

Implementacija QuickSort algoritma u AEC-u

Autor: Teo Samaržija

Sažetak: AEC je pojednostavljeni niski programski jezik koji je autor osmislio za potrebe komuniciranja sa svojim jednostavnim compilerom, koji je radio u nastojanju da bolje shvati kako funkcioniraju compileri. U ovom seminaru opisuje se autorov pokušaj da efikasno implementira QuickSort (quick na engleskom znači brz, a sort znači razvrstati), jedan od najčešće korištenih algoritama razvrstavanja prema nekoj poredbenoj funkciji, u tom jeziku. Također se opisuje i pokušaj da se tehnikom mekanog programiranja (točnije, genetskim algoritmom) dođe do formule koja će predviđati koliko će usporedbi QuickSort napraviti na nekom nizu prije no što ga razvrsta do kraja. Naivni genetski algoritam, nažalost, došao je do formule koja relativno dobro opisuje ponašanje QuickSorta na velikim nizovima (za koje mu je autor dao rezultate mjerenja), ali koja drastično precjenjuje broj usporedbi koje će QuickSort napraviti za male već razvrstane nizove.

O AEC-u

AEC (kratica od *Arithmetic Expression Compiler* – sastavljač aritmetičkih izraza) je, kao što piše u sažetku, pojednostavljeni niski (relativno blizak asemblerskom) programski jezik koji je autor osmislio za potrebe komuniciranja sa svojim jednostavnim compilerom. Osmišljen je da bude, prije svega, jednostavan za prevoditi na asemblerski jezik, a da ipak nema previše zbunjujuću sintaksu (jezik sličan LISP-u možda bi bio još lakšim za prevoditi na asemblerski jezik, jer ga je lakše parsirati, no autor smatra da LISP ima previše zbunjujuću sintaksu, barem za ljude koji s njom nisu uopće upoznati). Autor je napravio nekoliko odluka koje bi jezik učinile lakši za prevoditi na asemblerski jezik, ali koje bi znatno otežale proširivanje jezika: compiler zahtijeva da programer sam deklarira varijable u umetnutim isječcima asemblerskog koda, jezik podržava samo jedan tip podatka (32-bitni decimalni broj), nizovi se podataka tretiraju kao asemblerske funkcije, te parser podržava samo funkcije s do dva argumenta i funkcije koje nisu uprogramirane u parser mogu imati samo jedan argument.

O compileru za AEC

Compiler za AEC pisan je u JavaScriptu i C-u, povezanih preko Duktape radnog okvira. Duktape (engleski za *ljepljiva traka*, u smislu *radni okvir koji spaja dva veoma različita programska jezika*) je interpreter za JavaScript pisan u cijelosti u C-u, po standardu C99 (koji prihvaća svaki današnji compiler za C), dostupan pod MIT licencom. Dakle, compiler za AEC sastoji se od četiri datoteke koda: *aec.c*, *compiler.js*, *duktape.c* i *control.js*¹, od kojih je *aec.c*, *compiler.js* i *control.js* napisao sam autor, i ukupno te tri datoteke imaju oko 2'000 redaka koda. *Aec.c* sastoji se od potprograma za inicijaliziranje Duktapea, funkcije koja, na zahtjev od *compiler.js*, pretvara decimalne brojeve u heksadekadski² IEEE 754 zapis (pomoću C-ove poznate naredbe *sprintf*), potprograma koji čita sadržaj *.aec* datoteke (gdje je spremljen neki kod na jeziku AEC) i šalje ga do *control.js*, potprograma koji na zaslon ispisuje poruke o pogrešci koje mu šalje *compiler.js*, te potprograma koji zapisuje asemblerski kod koji mu šalju *control.js* i *compiler.js* u *.asm* datoteku. Funkcija za pretvorbu decimalnog broja u heksadekadski, na primjer, izgleda ovako:

```
static duk_ret_t getIEEE754(duk_context *ctx)
{
    static char polje[16];
```

1 Dostupni u ZIP-arhivi na <https://flatassembler.000webhostapp.com/Duktape.zip>

Detaljne upute kako ih compilirati dostupni su u ovom videu: <https://youtu.be/N2C1i8CW7Io>

2 Namjerno sam upotrijebio rjeđe korištenu riječ *heksadekadski* umjesto *heksadecimalni*, jer smatram da riječ *heksadecimalni* nije ispravna. Naime, to bi trebalo značiti koji se odnosi na bazu 16. No, *heksa* je grčka riječ za broj šest, a *deci* je latinski korijen sa značenjem deset. Reći *heksadecimalni* je kao da kažemo *sixnaest* ili *šestteen*. Treba reći *heksadekadski* jer je *deka* grčka riječ za broj deset, ili reći *sedecimalni*, budući da je *sedecim* latinska riječ za broj 16.

```
float broj=duk_to_number(ctx,0);
sprintf(polje,"0x%X",*(int*)&broj);
duk_push_string(ctx,polje);
return 1;
```

```
}
```

duk_ret_t (malim početnim slovom, važno je jer C razlikuje velika i mala slova) je cijeli broj koji označava koliko smo podataka vratili u JavaScript. duk_context (ponovno, malim početnim slovom, inače je to u C-u gramatički netočno) je JavaScriptina virtualna mašina. duk_to_number vraća podatak koji je na vrhu sistemskog stoga JavaScriptine virtualne mašine, pod pretpostavkom da je taj podatak broj (ako nije, program prestaje s radom s porukom o fatalnoj pogrešci). duk_push_string stavlja niz znakova na vrh sistemskog stoga te JavaScriptine virtualne mašine, tako da ga program koji se pokreće na njoj može čitati.³

Control.js sastoji se od jedne dugačke funkcije koja prima jednu liniju koda od *aec.c* (naravno, posredstvom *duktape.c-a*), ekstrapolira aritmetički izraz iz nje, šalje taj izraz do *compiler.js*, te obradi ostatak te linije. Recimo, ako linija AEC-ovskog koda glasi `If a<b`, tada se `a<b` šalje do *compiler.js*, a `If` se zasebno obrađuje. U JavaScriptu to izgleda ovako:

```
if (/^If/.test(str))
{
    var arth=str.substr("If ".length);
    parseArth(tokenizeArth(arth)).compile();
    asm("fistp dword [result]");
    asm("mov eax,[result]");
    asm("test eax,eax");
    var label1="1"+(Math.floor(Math.random()*1000000));
    var label2="1"+(Math.floor(Math.random()*1000000));
    stack.push(label2);
    stack.push(label1);
    asm("jz "+label1);
    hasElse.push(false);
}
```

`/^If/` je regularni izraz koji prihvaća svaki niz znakova čiji je prvi znak 'I' (veliko slovo 'i'), a drugi znak malo slovo 'f'. `parseArth` (da, i JavaScript razlikuje velika i mala slova, i bilo bi pogrešno napisati `ParseArth`) i `tokenizeArth` su potprogrami koje je autor napisao u datoteci *compiler.js*. `tokenizeArth` prima od *control.js* niz znakova koji predstavljaju aritmetički izraz, a potprogramu koji ga je pozvao (upravo *control.js-u*) vraća niz sastavljen od nizova znakova. Zatim *control.js* šalje taj niz nizova znakova potprogramu `parseArth`. Ovakvu skraćenu gramatičku konstrukciju omogućuje to što su potprogrami `parseArth` i `tokenizeArth` u *compiler.js* deklarirani kao funkcije. Glavni razlog zašto sam ih deklarirao kao funkcije, a ne kao module, je to što sam često koristio zastarjele ili namjerno ograničene interpretere za JavaScript, a mnogi od njih ne podržavaju module (ni Internet Explorer ni Duktape ih ne podržavaju). `compile` je metoda koju imaju objekti koje stvara *prototipna funkcija* `Token`, a koje pozivatelju vraća `parseArth`. *Metoda* i *prototipna funkcija* su termini iz naprednog objektivnog orijentiranog programiranja, nije važno ako ne znate što znače. U slučaju da to nekog utješi, starije verzije JavaScripta (a značajke novijih verzija JavaScripta nisu korištene u ovom programu da se ne naleti na probleme s kompatibilnosti) nisu imale klase, nego se u njima objektivno orijentirano programiranje radilo na malo drukčiji način nego u drugim jezicima. *Prototipna funkcija* približno odgovara konstruktoru u drugim jezicima. `asm` je potprogram koji preko Duktapea do *aec.c* šalje niz znakova (ako je sve u redu, na gramatički točnom i smislenom asemblerskom jeziku) koji se treba napisati u datoteku *.asm*, a potprogramu koji ga je pozvao ne šalje ništa.

³ Izvor: <https://duktape.org/api.html>

Jezgra kompilera je, naravno, datoteka *compiler.js*. U nesažetom obliku ona sadrži nešto manje od 1700 redaka. Da opišem kako kod u njoj funkcionira, vjerojatno je najjednostavnije da je podijelim na četiri dijela: dio uglavnom neovisan o okruženju u kojem se vrti, dio koji pretpostavlja da okruženje u kojem se vrti može interpretirati osnovni HTML, dio koji pretpostavlja da okruženje u kojem se vrti podržava i osnovni HTML i osnovni DOM (*Document Object Model – objektivni model dokumenta* - protokol kojim programi kao što su Trident, ugrađen u Internet Explorer, Gecko, ugrađen u Firefox, Blink, ugrađen u Chrome, ili WebKit, ugrađen u Safari i Android Stock Browser, komuniciraju s programima pisanim u JavaScriptu), te dio koji pretpostavlja da okruženje u kojem se pokreće podržava napredni HTML, relativno napredni DOM i osnovni SVG (*Scalable Vector Graphics – vektorska grafika promjenjive veličine* - protokol kojim JavaScriptski programi mogu reći Tridentu i sličnim programima da crtaju vektorsku grafiku). Ovdje u Duktapeu samo se dio koji je uglavnom neovisan o okruženju u kojem se vrti može vrtjeti. To uključuje već spomenute potprograme `tokenizeArth`, `tokenizer`, `parseArth`, `parser`, te metode koje imaju objekti koje stvara funkcija `Token`, uključujući `compile`, po kojoj se ovaj program i zove `compiler`, te `interpret`, evaluator aritmetičkih izraza koje je `parseArth` već parsirao. Kažem da je uglavnom neovisan, jer on na jednom mjestu pita okruženje u kojem se vrti (Duktape, SpiderMonkey, V8, Chakra, Rhino...) zna li što znači `getIEEE754`, rečenicom `if (typeof getIEEE754 != "function")`. To je zapravo riječ koju je autor ovog teksta izmislio, i JavaScriptinom okruženju što to znači treba objasniti neki potprogram koji nije pisan u JavaScriptu, kao što je gore napravljeno u C-u. Compiler za AEC cilja ponajprije da u *.asm* ispisuje kod koji je kompatibilan s compilerom za asemblerski jezik koji se zove FlatAssembler. No, on isto tako cilja i da se asemblerski kod koji on ispiše lako prebaci na neki drugi asemblerski compiler. FlatAssembler može sam pretvarati decimalne brojeve u IEEE754 zapis, kakav se nalazi u strojnom jeziku (nule i jedinice koje računalno može razumjeti bez pomoći dodatnih programa). No, većina kompilera za asemblerski jezik ne zna sama pretvarati decimalne brojeve u IEEE754. Također, nije očito kako pretvoriti decimalni broj u IEEE754 zapis u starijim verzijama JavaScripta, a da se dobije najveća preciznost koju 32-bitni IEEE754 zapis dopušta. No, očito je kako to napraviti u C-u, i u brojnim drugim jezicima. Vjerojatno je moguće to napraviti u modernom JavaScriptu koristeći nizne sabirnice (`ArrayBuffer`), no ni to nije baš očito kako.

Tokenizer je dio kompilera koji razdvaja riječi iz programskog jezika, koji ostalim dijelovima kompilera označava gdje završava koja riječ. Kao primjer kako funkcionira taj moj tokenizer, dat ću dio programa koji provjerava jesu li brojevi ispravno napisani (sadrži li možda decimalni broj dvije decimalne točke...):

```
if (ret[i].charAt(0) >= '0' && ret[i].charAt(0) <= '9')
{
    var decimal = 0;
    for (var j = 0; j < ret[i].length; j++)
    {
        if (decimal && ret[i].charAt(j) == '.' ||
(ret[i].charAt(j) < '0' || ret[i].charAt(j) > '9') &&
ret[i].charAt(j) != '.')
        {
            alert("Tokenizer error: Can't assign the type to
the token \"" + ret[i] + "\".");
            alerted = 1;
            return ret;
        } else if (!decimal && ret[i].charAt(j) == '.')
            decimal = 1;
    }
}
```

Ovdje `alert`, ukoliko se program vrti u internetskom pregledniku, prikazuje skočni prozor s porukom o pogrešci. Inače, on baca niz znakova koji mu je dan kao argument (između okruglih

zagrada '(' i ')') kao iznimku (engleski *exception*, način na koji se, u objektivno orijentiranom programiranju, potprogrami međusobno obavještavaju o pogreškama). U Duktapeu ta se se iznimka onda, kako je *compiler.js* nigdje ne hvata, šalje u *control.js*, pa zatim u Duktape, pa zatim Duktape poziva C funkciju iz *aec.c* koja mu je zadana za ispisivanje poruka o pogrešci. Ta funkcija također ispisuje i redak programa pisanog u AEC-u gdje je do pogreške došlo. Smatram da se iz ovoga može shvatiti kako funkcionira tokenizer tog kompilera za AEC.

Parser je dio kompilera koji ostalim dijelovima kompilera označava koja je riječ u programskom jeziku sintaksno povezana s kojom. To se radi pravljjenjem sintaksnih stabala u memoriji. Većina kompilera, još od doba prvih viših programskih jezika, za parsiranje aritmetičkih i logičkih izraza koristi Dijkstrin Shunting-Yard algoritam. Njegova prednost je to što se vrti u linearnom vremenu. Compiler za AEC ga ne upotrebljava, nego je autor osmislio svoj algoritam koji se lako iskaže u JavaScriptu. On se vrti u kvadratnom vremenu ako nema zagrada, te u kubnom vremenu ako ima zagrada. Iako je to u teoriji mnogo lošije od Shunting Yarda, u praksi se na današnjim računalima čak i najduži izrazi koji se u praksi nalaze u programima kompiliraju u zanemarivo kratkom vremenu. Kao primjer kako funkcionira parser u kompileru za AEC, evo koda koji radi s logičkim operatorima & (logičko I) i | (logičko ILI):

```
for (var i = 0; i < arth.length; i++)
    if ((arth[i].text == '&') && !arth[i].operands.length)
    {
        if (arth.length - 1 == i || !i)
        {
            alerted = 1;
            alert("Parser error: The binary operator '\" +
arth[i].text + "\" has less than two operands.");
            return arth[0];
        }
        arth[i].operands.push(arth[i - 1]);
        arth[i].operands.push(arth[i + 1]);
        arth.splice(i - 1, 1);
        arth.splice(i, 1);
        i--;
    }
for (var i = 0; i < arth.length; i++)
    if ((arth[i].text == '|') && !arth[i].operands.length)
    {
        if (arth.length - 1 == i || !i)
        {
            alerted = 1;
            alert("Parser error: The binary operator '\" +
arth[i].text + "\" has less than two operands.");
            return arth[0];
        }
        arth[i].operands.push(arth[i - 1]);
        arth[i].operands.push(arth[i + 1]);
        arth.splice(i - 1, 1);
        arth.splice(i, 1);
        i--;
    }
}
```

for-petlja koja traži &-e izvršava se prije for-petlje koja traži |-e jer logički operator & u AEC-u, kao i u većini programskih jezika, ima veći prioritet od logičkog operatora |. Gotovo identičan algoritam je za aritmetičke operatore. Dakle, taj program opet i opet prolazi kroz niz Token-a zvan

arth. Kada se zadovolje uvjeti za to, prethodni i sljedeći element tog niza dodaju se u polje `operands`, koje je svojstvo (engleski *property*, u objektivno orijentiranom programiranju, to je objekt ili varijabla koji se nalaze u drugom objektu) Token-a (prije tih `for-petlji`, `operands` je za svaki element tog polja prazan niz), te se JavaScript naredbom `splice` brišu iz niza `arth`. Ako je sve u redu (nema *unexpectednih tokena*, neočekivanih oznaka), na kraju parsiranja u `arth`-u je samo jedan Token, koji je onda korijen sintaksnog stabla. Ako je tako, `parseArth` šalje potprogramu koji ga je pozvao prvi element niza `arth`, inače javlja grešku *unexpected token*.

Jedna od zanimljivosti o AEC-u je da u njemu mogu postojati gramatičke iluzije, rečenice koje se mogu parsirati, a da su zapravo sintakсно još uvijek netočne i nemaju značenje. Poznati primjer za to u ljudskim jezicima je rečenica *More people have been to Russia than I have*. – zvuči prihvatljivo kad je čujemo, ali kada razmislimo što bi mogla značiti, vidimo da se ne može niti gramatički analizirati. Zato, nakon što je parser završio s radom, poziva se sljedeća funkcija koja se pobrine za gramatičke iluzije:

```
function checkAST(node) //Postoje li "gramaticke iluzije", izrazi
koji se mogu parsirati, ali ne znace nista (recimo "5**").
{
    if ((node.text == '*' || node.text == '-' || node.text == '/'
    || node.text == '=' || node.text == '<' || node.text == '>'
    || node.text == '&' || node.text == '|')
    && node.operands.length < 2 && !alerted)
    {
        alerted = 1;
        alert("Parser error: Unexpected token \"'\" + node.text +
        "\"\'.");
    }
    for (var i = 0; i < node.operands.length; i++)
        checkAST(node.operands[i]);
}
```

Kako je FPU arhitektura bazirana na stogu, relativno je lagano compilirati AST (sintakсно stablo) u njezin assembler. U biti, compiler pokrene rekurziju u korijenu AST-a, a ta rekurzija prvo pokrene rekurziju u svakom potomku (elementu polja `operands`) Token-a, i zatim šalje poruku dijelu programa koji ispisuje *.asm* datoteku s prijevodom svog *text*-a na assembler. Mali problem je to što FPU-ov stog može držati samo 8 brojeva, pa će naivna implementacija takvog algoritma u asemblerskom programu za duže izraze vjerojatno uzrokovati *stack overflow*. Rješenje je prvo pozvati rekurziju za Token u polju `operands` s manje razina potomaka, a koliko koji Token u stablu ima razina potomaka, određuje se DFS algoritmom s memoizacijom:

```
var zamijeni = false;
if (this.operands.length > 1 && this.operands[0].DFS() <
this.operands[1].DFS())
{
    zamijeni = true;
    var tmp = this.operands[0];
    this.operands[0] = this.operands[1];
    this.operands[1] = tmp;
}
for (var i = 0; i < this.operands.length; i++)
    this.operands[i].compile();
if (zamijeni)
    asm("fexch");
```

Gdje je DFS definiran kao:

```

ret.DFS = function () // Pokusat cu izbjeći stack overflow tako da
se prvo kompajliraju "dublji" izrazi.
{
    if (this.depth)
        return this.depth;
    if (!this.operands.length)
        this.depth = 1;
    else
        for (var i = 0; i < this.operands.length; i++)
            this.depth = Math.max(this.depth,
this.operands[i].DFS() + 1);
    return this.depth;
}

```

Nakon toga slijedi niz else-if-ova kao što su:

```

else if (this.text == "sqrt(")
    fsqrt();
else if (this.text == "arcsin(")
    fasinp();
else if (this.text == "arccos(")
    facosp();

```

Potprogrami koji se ovdje spominju imaju svoje definicije, nekad jednostavne, a nekad relativno složene. Recimo, facosp je definiran kao:

```

function facosp() // -||- arkus kosinus (po formuli pi/2-
arcsin(x))
{
    asm("fstp dword [result]");
    asm("fldpi");
    asm("fld1");
    asm("fld1");
    faddp();
    fdivp();
    asm("fld dword [result]");
    fasinp();
    fsubp();
}

```

Evo, nadam se da sam objasnio kako funkcionira jezgra kompilera za AEC. Interpretiranje aritmetičkih izraza je trivijalno, samo se u else-if-ovima grana rekurzija:

```

else if (this.text == "+")
    return this.operands[0].interpret() +
this.operands[1].interpret();
else if (this.text == "-")
    return this.operands[0].interpret() -
this.operands[1].interpret();

```

No, kao što sam već spomenuo, to je samo dio onoga što se nalazi u *compiler.js*. U *compiler.js* nalazi se i dio koji će se vrtjeti u okruženju koje podržava osnovni HTML, ali ne i osnovni DOM. To je dio za sintaksno bojanje koda.

Datoteku *compiler.js* koristio sam i u programu Simple Calculator⁴, tamo sam u programskom jeziku Java stvorio okruženje od Mozillinog radnog okvira Rhino (za *just-in-time* prevođenje JavaScripta u binarni kod kompatibilan s Java virtualnom mašinom) i Oracleovog radnog okvira za aplikacije s grafičkim sučeljem Swing. Swing podržava osnovni HTML, ali ne i DOM. Ipak, kod za sintaksno bojanje asemblerskog koda može se vrtjeti u takvom okruženju.

Zatim postoji dio *compiler.js*-a koji se vrti ako se pokrene u okruženju koji, osim osnovnog HTML-a, podržava i osnovni DOM. On namješta kako web-stranica tog mog kompilera⁵ treba izgledati kada se pokrene u internetskom pregledniku. Izradio sam da izgleda kao aplikacija za starije verzije



Illustration 1: Sintaksno bojanje asemblerskog koda funkcioniра u okruženju od radnih okvira Rhino i Swing, iako ni jedno od njih ne podržava DOM, zato što Swing podržava HTML

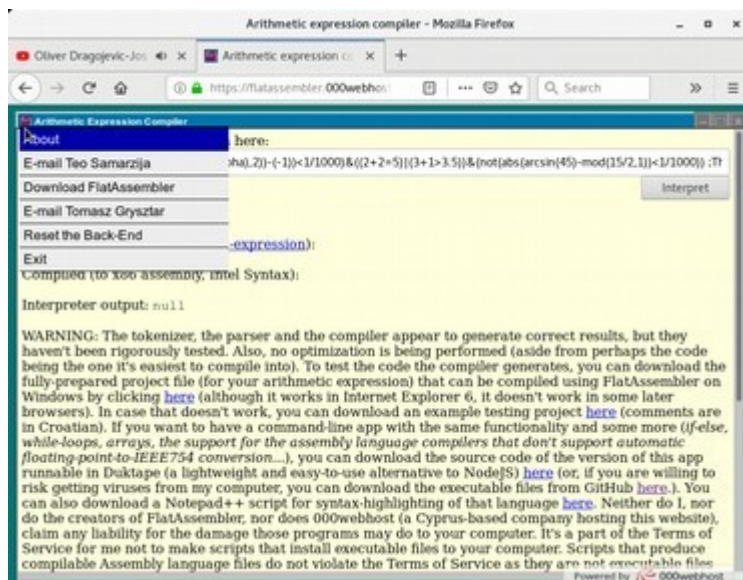


Illustration 2: Kada se pokrene u internetskom pregledniku, compiler za AEC izgleda kao aplikacija na starijim verzijama Windowsa

Taj dio programa pretpostavlja da je okruženje u kojem se vrti internetski preglednik, ali radi minimalne pretpostavke o tome što taj internetski preglednik podržava, do te mjere da radi i u Internet Exploreru 6.

Programski kod u *compiler.js* ima i dio koji se vrti samo u modernim internetskim preglednicima, to jest koji podržavaju napredni HTML i DOM te barem osnove SVG-a. Taj dio koda crta AST-ove pomoću SVG-a. U starijim je internetskim preglednicima jedini način prikazivanja AST-a pretvaranje AST-a u LISP-ove S-izraze, koji su početnicima u programiranju vjerojatno još više zbunjujući nego aritmetički izrazi kojima su dodane zagrade. Grafička reprezentacija jednostavnih AST-ova kakvu daje compiler za AEC u modernim internetskim preglednicima vjerojatno može pomoći početnicima u programiranju da shvate koncept AST-a, a samim time i parsiranja.

⁴ Možete ga skinuti s GitHuba, velik je približno 1 MB:

<https://github.com/FlatAssembler/SimpleCalculator/raw/master/SimpleCalculator.zip>

⁵ <https://flatassembler.000webhostapp.com/compiler.html>

Internetski preglednik koji je dovoljno moderan da se taj potprogram za crtanje AST-a pokrene danas se može staviti na gotovo svako računalo ili mobitel, on radi u Internet Exploreru 11. Iako neka računala ili mobiteli još uvijek imaju manje moćne internetske preglednike, gotovo bi svaki mogao pokrenuti dovoljno moderan preglednik kada bi se instalirao na njega.

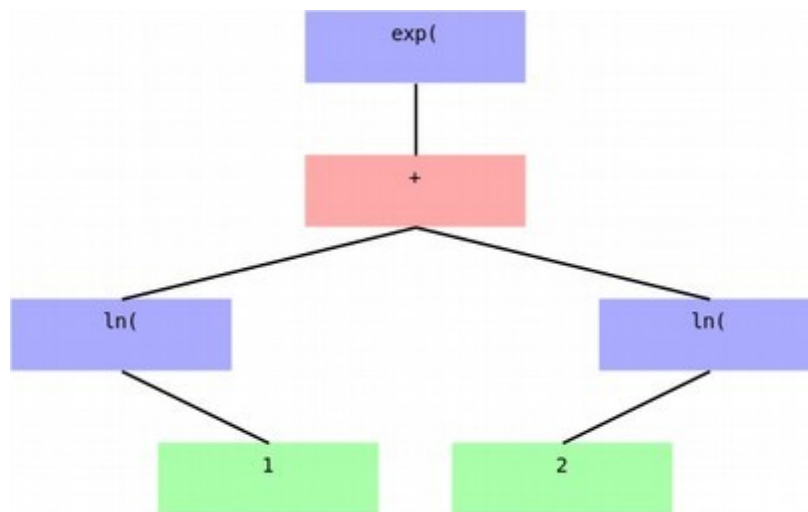


Illustration 3: Sintaksno stablo (AST) koje je compiler nacrtao za izraz $\exp(\ln(1)+\ln(2))$

Objašnjenje programa za razvrstavanje pisanog u AEC-u

Krenimo, konačno, na objašnjenje programa za razvrstavanje pisanog na programskom jeziku AEC⁶. Autor ovog teksta napisao je skriptu za Notepad++ koja sintaksno oboji kod pisan u AEC-u⁷, te je naredio Notepad++-u da ga izveze u RTF datoteku (koja se može otvoriti u LibreOfficeu) zajedno s brojevima koje označuju linije koda, tako da će ovdje biti moguće umetnuti sintaksno obojeni AEC.

```

1.      ;Implementacija QuickSort-a.
2.      AsmStart ;Umetnuti Assembler pocinje ovako, a završava sa "AsmEnd".
3.      ispisPoruka=1 ;Ovako se rade pretprocesorski definesovi u
FlatAssembleru.
4.      debug=0
5.      macro staviIntNaSistemskiStog x ;Da, ima on mocan
pretprocesor.
6.      {
7.          sub esp,4
8.          fld dword [x]
9.          fistp dword [esp]
10.     }
11.     macro staviPokazivacNaSistemskiStog x
12.     {
13.         sub esp,4
14.         lea ebx,[x]
15.         mov [esp],ebx
16.     }
17.     macro staviStringNaSistemskiStog x

```

⁶ Izvršna datoteka, velika 6 KB, dostupna je na:

<https://github.com/FlatAssembler/ArithmeticExpressionCompiler/raw/master/QuickSort/qsor.exe>

⁷ Skripta pisana na jeziku XML koja će naučiti Notepad++ da sintaksno oboji tekst pisan na AEC-u dostupna je ovdje: <https://flatassembler.000webhostapp.com/Arth.xml>


```

18.      {
19.          sub esp,4
20.          mov dword [esp],x
21.      }

```

U ovom programu pozivat ću mnoge naredbe iz C-a. Za to ću koristiti umetnuti Assembler. Kako je pozivati C-ove naredbe iz asemblerskog koda ponavljajuće i sklono greškama, napravio sam par FlatAssemblerskih makro-naredbi koje će mi u tome pomoći. StaviIntNaSistemiStog (u 5. retku) je možda malo neprimjeren naziv, jer ono što ta makronaredba zapravo čini jest pretvoriti 32-bitni decimalni broj u 32-bitni cijeli broj (C-ov `int`), pa ga tek onda stavlja na sistemski stog. Komentari (dijelovi teksta na programskom jeziku koje compiler preskače, najčešće su ti dijelovi teksta zapravo pisani na engleskom jeziku, ovdje su pisani na hrvatskom), kao što vidite, i u FlatAssembleru i u AEC-u pišu se između znaka ';' (točka-zarez) i kraja retka.

```

22.      format PE console ; 'PE' je 32-bitna '.EXE' datoteka za
Windows. 'PE64' je 64-bitna '.EXE' za Windows. 'MZ' je '.EXE' za DOS. 'ELF'
je izvršna datoteka za 32-bitni Linux, a 'ELF64' za 64-bitni.
23.      entry start
24.
25.      include 'win32a.inc' ; Naredbe za komunikaciju s DLL-ovima.
26.
27.      section '.text' code executable
28.      start:

```

Dio koda od 22. do 28. linije ilustrira veliku razliku između AEC-a i drugih niskih programskih jezika: AEC očekuje da programer komunicira s asemblerskim compilerom o tome kako formatirati izvršnu datoteku. AEC compiler ne radi pretpostavke o tome kako će izvršna datoteka biti formatirana. C compileri to uvelike rade... i u tome nerijetko griješe, pogotovo kad programer želi raditi egzotične vrste izvršnih datoteka, kakve je znatno lakše raditi u asemblerskom jeziku nego u C-u.

```

29.      if ispisPoruka=1 ; 'if' je ovdje assemblerska pretprocesorska
naredba. 'If', s velikim 'i', je naredba grananja u AEC-u.
30.      jmp velicinaUnosa$
31.      velicinaUnosa db "Unesite koliko cete brojeva
unijeti.",10,0
32.      velicinaUnosa$:

```

Ovo sigurno izgleda čudno nekome tko je upoznat s višim programskim jezicima, ali ne i s asemblerskim. Da, asemblerski jezik dopušta da u izvršni dio programa umetnete nizove znakova (koje su, naravno, nedopuštene instrukcije za procesor) i sve će biti u redu dokle god ih preskočite (recimo, naredbom `jmp`). I, da, možete potprogramima izvan vašeg programa (recimo, C-ovim naredbama) slati pokazivače na te nizove znakova, a da se program ne sruši. Ono 10,0 na kraju 31. retka označava znak za novi red i znak za završetak stringa (niza znakova), zato jer je mjestu 10 u ASCII tablici znak za novi red, a C-ove naredbe očekuju da niz znakova koji im se pošalje završava znakom koji je na mjestu 0 u ASCII tablici.

```

33.      staviStringNaSistemiStog velicinaUnosa
34.      call [printf]

```

I ovo je primjer kako se pozivaju C-ove naredbe (u ovom slučaju `printf`) iz asemblerskog koda. U biti, trebamo se pobrinuti da `esp` (procesorski registar koji, u biti, pokazuje neko mjesto na sistemskom stogu) pokazuje na prvi argument, te da odmah nakon prvog argumenta (recimo, ako je prvi argument velik 4 bajta, onda na memorijskoj adresi `esp+4`) stiže drugi argument, i tako dalje. Barem je tako na Windowsima, na Linuxu i MacOS-u je znatno kompliciranije (tamo C-ove naredbe očekuju prvih nekoliko argumenata u procesorskim registrima).

```

35.      end if
36.      staviPokazivacNaSistemiStog n
37.      jmp znakZaFloat$

```

```

38.          znakZaFloat db "%f",0
39.          znakZaFloat$:
40.          staviStringNaSistemskiStog znakZaFloat
41.          call [scanf]
42.          if ispisPoruka=1
43.              jmp pitajZaUnos$
44.              pitajZaUnos db "Unesite te brojeve:",10,0
45.              pitajZaUnos$:
46.              staviStringNaSistemskiStog pitajZaUnos
47.              call [printf]
48.          end if
49.      AsmEnd
50.      i:=0
51.      brojac:=0
52.      vrhStoga:=0
53.      While i<n
54.          pokazivac:=4*i
55.          AsmStart
56.              fld dword [pokazivac]
57.              fistp dword [pokazivac]
58.              lea ebx,[original]
59.              add ebx,[pokazivac]
60.              staviPokazivacNaSistemskiStog ebx
61.              staviStringNaSistemskiStog znakZaFloat
62.              call [scanf]
63.          AsmEnd
64.          i:=i+1
65.      EndWhile
66.      AsmStart
67.          call [clock]
68.          mov [procesorskoVrijeme],eax

```

C-ova naredba `clock` (pozvana u 67. retku) trebala bi vratiti 64-bitni cijeli broj, i problem je s time upravljati u asemblerskom kodu. No, na sreću, ona zadnja 32 bita vraća upravo u `eax`, gdje i funkcije koje vraćaju 32-bitne cijele brojeve vraćaju svoj rezultat, tako da se o tome ne moramo brinuti. Inače, `clock` na Windowsima programu koji je pozove vraća broj milisekundi (jedna sekunda ima 1000 milisekundi) koje su prošle otkad se taj program počeo vrtjeti.

```

69.      AsmEnd
70.      razvrstanost:=0
71.      i:=0
72.      While i<n-1
73.          razvrstanost:=razvrstanost+(original(i)<original(i+1))
74.          i:=i+1
75.      EndWhile
76.      razvrstanost:=razvrstanost/((n-1)/2)-1

```

Dakle, ovdje mjerimo do koje mjere je niz koji smo unijeli već razvrstan. To je važno da vidimo koliko je formula koju je izveo genetski algoritam za predviđanje broja usporedbi koje će QuickSort napraviti točna, više o tome kasnije.

```

77.      i:=2
78.      While i<7 | i=7 ;Kada nisam stavio operator "<=" u svoj jezik.
79.          razvrstanostNa(i):=pow(abs(razvrstanost),i) ;Zato sto je

```

"pow(x,y)" u tom mom jeziku samo sintakticki secer za "exp(ln(x)*y)", i to vraća "NaN" za x<=0. Nema ocitog nacina da se "pow" prevede na Assembler.

```

80.         If razvrstanost=0
81.             razvrstanostNa(i) := 0
82.         EndIf
83.         If mod(i,2)=1 & razvrstanost<0
84.             razvrstanostNa(i) := -razvrstanostNa(i)
85.         EndIf
86.         i:=i+1
87.     EndWhile
88.     ;f(n,s)=exp((ln(n)+ln(ln(n)))*1.05+(ln(n)-
ln(ln(n)))*0.83*abs(2.38854*pow(s,7)-0.284258*pow(s,6)-
1.87104*pow(s,5)+0.372637*pow(s,4)+0.167242*pow(s,3)-
0.0884977*pow(s,2)+0.315119*s))

```

I u 88. retku imate tu formulu, kasnije će pisati mnogo više o njoj. U biti, ovaj dio programa je dodan naknadno.

```

89.     polinomPodApsolutnom:=2.38854*razvrstanostNa(7)-
0.284258*razvrstanostNa(6)-
1.87104*razvrstanostNa(5)+0.372637*razvrstanostNa(4)+0.167242*razvrstanostNa(
3)-0.0884977*razvrstanostNa(2)+0.315119*razvrstanost
90.     eNaKoju:=(ln(n)+ln(ln(n)))*1.05+(ln(n)-
ln(ln(n)))*0.83*abs(polinomPodApsolutnom)
91.     ocekivaniBrojUsporedbi:=exp(eNaKoju)
92.     najmanjiCijeliBrojKojiSeMozeDodatiNaBrojac:=1

```

Ovo u 92. retku važno je zato što AEC podržava samo jedan tip podatka, 32-bitni decimalni broj. On može prikazati mnogo veće i mnogo manje brojeve nego što može 32-bitni cijeli broj, no zato može prikazivati samo otprilike prvih 7 znamenki tih brojeva. Zato kada izbrojimo do nešto manje od 17 milijuna, dodavanje jedinice na brojač više nema efekta. No, dodavanje dvojke još će jedno vrijeme imati efekta. Nakon što ni dodavanje dvojke ne bude imalo efekta, još će jedno vrijeme dodavanje četvorke imati efekta, i tako dalje. Trenutak kada dodavanje broja koji dodajemo više nema efekta, zove se preljev (engleski *overflow*).

```

93.     pomocniBrojac:=0

```

Pomoćni brojač je važan jer, kada jednom dodavanje jedinice više ne bude imalo efekta, ne smijemo dodati dvojku svaki put kad se izvrši petlja čiji broj izvršavanja želimo izbrojati, već svaki drugi put.

Sada slijedi glavni dio programa, implementacija QuickSort algoritma. QuickSort algoritam bazira se na činjenici da je moguće u linearnom vremenu preurediti niz tako da se svi elementi manji od onog koji je prije bio prvi (pivot) stave prije njega, a svi veći ili jednaki njemu nakon njega. Ima nekoliko načina da se to učini, a najjednostavniji je najvjerojatnije pomoću pomoćnog niza. Trik je u tome da, kada smo to napravili za jedan element, koji je sada došao negdje na sredinu niza, ne moramo opet proći kroz čitav niz da bismo to napravili za neki drugi element, nego samo kroz približno polovicu niza. Naime, kada nakon toga brojimo koliko elemenata ima u nizu koji su manji od elementa koji je sada prvi, ne moramo brojati dalje od pivoata (onog elementa koji je prije bio prvi), jer elementi nakon pivoata ne mogu više biti manje od elementa koji je sada prvi u nizu. Tako se broj izvršavanja unutrašnje petlje stalno raspolavlja, a, naravno, ako se stalno raspolavlja, potrebno mu je $\log_2(n)$ raspolavljanja da dođe do jedinice. Na taj se način QuickSort izvršava u linearnitičnom vremenu $n \cdot \log_2(n)$. To je mnogo brže od kvadratnog vremena za koje se vrte naivni algoritmi razvrstavanja elemenata po veličini, kao što je odabirno razvrstavanje (SelectionSort).

SelectionSort je mnogo sporiji od QuickSorta ukoliko je uspoređivanje dva elementa spora operacija, recimo, uspoređivanje koji niz znakova ide prije po ASCII abecedi (čest problem u programiranju). No, on je ipak brži od QuickSorta ukoliko je zamjena dva elementa mnogo sporija od usporedbe dva elementa, jer SelectionSort uvijek radi samo nužne zamjene. Recimo, kada pisanjem olovkom i brisanjem gumicom želimo poredati niz napisan olovkom na papiru, to ćemo brže napraviti držimo li se SelectionSort algoritma nego QuickSort algoritma. U praksi se takve situacije u računalima ne događaju, i za razvrstavanje velike količine podataka na sporom tvrdom

disku uglavnom se koristi ljuskasto razvrstavanje (ShellSort), koje se vrti u hiperlinearitmičnom vremenu $n \cdot \log_2(n) \cdot \log_2(n)$, gdje je \log_2 binarni logaritam.

```
94.   vrhStoga:=vrhStoga+1
95.   stogSDonjimGranicama(vrhStoga):=0
96.   stogSGornjimGranicama(vrhStoga):=n
97.   While vrhStoga>0
98.       gornjaGranica:=stogSGornjimGranicama(vrhStoga)
99.       donjaGranica:=stogSDonjimGranicama(vrhStoga)
100.      vrhStoga:=vrhStoga-1
101.      gdjeJePivot:=donjaGranica
102.      i:=donjaGranica+1
103.      While i<gornjaGranica
104.          If original(i)<original(donjaGranica)
105.              gdjeJePivot:=gdjeJePivot+1
106.          EndIf
107.          i:=i++
108.      EndWhile
109.      staviManje:=donjaGranica
110.      staviVece:=gdjeJePivot+1
111.      pomocni(gdjeJePivot):=original(donjaGranica)
112.      i:=donjaGranica+1
113.      While i<gornjaGranica
114.          If original(i)<original(donjaGranica)
115.              pomocni(staviManje):=original(i)
116.              staviManje:=staviManje+1
117.          Else
118.              pomocni(staviVece):=original(i)
119.              staviVece:=staviVece+1
120.          EndIf
121.          pomocniBrojac:=pomocniBrojac+1
122.          If
pomocniBrojac=najmanjiCijeliBrojKojiSeMozeDodatiNaBrojac
123.              brojac:=brojac+pomocniBrojac
124.              pomocniBrojac:=0
125.          EndIf
126.          i:=i+1
127.      EndWhile
128.      i:=donjaGranica
129.      While i<gornjaGranica
130.          original(i):=pomocni(i)
131.          i:=i+1
132.      EndWhile
133.      If gdjeJePivot<gornjaGranica-1
134.          vrhStoga:=vrhStoga+1
135.          stogSDonjimGranicama(vrhStoga):=gdjeJePivot+1
136.          stogSGornjimGranicama(vrhStoga):=gornjaGranica
137.      EndIf
138.      If gdjeJePivot>donjaGranica+1
139.          vrhStoga:=vrhStoga+1
140.          stogSDonjimGranicama(vrhStoga):=donjaGranica
141.          stogSGornjimGranicama(vrhStoga):=gdjeJePivot
```

```

142.         EndIf
143.     testZaPreljev:=brojac+najmanjiCijeliBrojKojiSeMozeDodatiNaBrojac
;Potrebna je posebna varijabla za to jer FPU interno radi s 80-bitnim
brojevima, a CPU s 32-bitnim.

```

To jest, kod kao što je `brojac + najmanjiCijeliBrojKojiSeMozeDodatiNaBrojac = brojac` ne bi funkcionirao za detekciju preljeva (overflowa).

```

144.         If not(testZaPreljev>brojac)
145.

```

```

najmanjiCijeliBrojKojiSeMozeDodatiNaBrojac:=najmanjiCijeliBrojKojiSeMozeDodatiNaBrojac*2

```

```

146.         AsmStart
147.             if ispisPoruka=1
148.                 jmp izvjesceOpreljevu$
149.                 izvjesceOpreljevu db "Upozorenje:
Brojac mozda nece sadrzavati tocan rezultat, dogodio se preljev na %d.
iteraciji."
150.                                     db " Najveca ocekivana pogreska za
ovaj preljev je %d krivo prebrojanih izvrsavanja unutarnje petlje.",10,0
151.                 izvjesceOpreljevu$:
152.                 fld dword [n]
153.                 fld dword
[najmanjiCijeliBrojKojiSeMozeDodatiNaBrojac]
154.                 fsubp
155.                 fabs
156.                 fistp dword [esp+4]
157.                 fld dword [brojac]
158.                 fistp dword [esp]
159.                 invoke printf,izvjesceOpreljevu
160.             end if
161.         AsmEnd
162.     EndIf
163. EndWhile

```

I ovdje završava QuickSort algoritam, sad slijedi ispisivanje rezultata.

```

164. AsmStart
165.     call [clock]
166.     sub eax,[procesorskoVrijeme]
167.     mov [procesorskoVrijeme],eax
168. if ispisPoruka=1
169.     jmp sortiraniNizJe$
170.     sortiraniNizJe db "Sortirani niz je:",10,0
171.     sortiraniNizJe$:
172.     staviStringNaSistemiStog sortiraniNizJe
173.     call [printf]
174. end if
175. AsmEnd
176. i:=0
177. While i<n
178.     pokazivac:=4*i
179.     AsmStart
180.         lea ebx,[original]
181.         fld dword [pokazivac]

```



```

182.          fistp dword [pokazivac]
183.          add ebx,[pokazivac]
184.          fld dword [ebx]
185.          fstp qword [esp]
186.          staviStringNaSistemiStog
znakZaFloatPlusNoviRedPlusNulZnak
187.          call [printf]
188.          AsmEnd
189.          i:=i+1
190.      EndWhile
191.      AsmStart
192.      if ispisPoruka=1
193.          staviIntNaSistemiStog brojac
194.          staviStringNaSistemiStog untrasnjaPetljaString
195.          call [printf]
196.      AsmEnd
197.          brojac:=n*ln(n)/ln(2)
198.      AsmStart
199.          fld dword [brojac]
200.          fstp qword [esp]
201.          staviStringNaSistemiStog slozenostString
202.          call [printf]
203.          push dword [procesorskoVrijeme]
204.          invoke printf,sortiranjeJeTrajalo
205.          fld dword [razvrstanost]
206.          fstp qword [esp]
207.          invoke printf,stringORazvrstanosti
208.          fld dword [ocekivaniBrojUsporedbi]
209.          fstp qword [esp+8] ;Zato sto "printf" iz MSVCRT-a za "%f"
ocekuje 8-bajtni "double", i izgleda da ju nije moguće namjestiti da ocekuje
4-bajtni "float".
210.          fld dword [eNaKoku]
211.          fstp qword [esp]
212.          invoke printf,izvjestajOFormuli
213.          fld dword [polinomPodApsolutnom]
214.          fstp qword [esp]
215.          invoke printf,izvjestajOPolinomu
216.          invoke system,_pause ;"Press any key to continue..."
217.      end if
218.      invoke exit,0

```

Ako programirate na asemblerskom jeziku, morate pozvati neku naredbu koja će vaš program legalno dovršiti (recimo, C-ovu naredbu `exit`, pozvanu u 218. redu), inače će procesor kada dođe do kraja izvršnog dijela vašeg programa početi učitavati nedopuštene binarne naredbe.

```

219.
220. ;Deklaracije konstanti.
221. _pause db "PAUSE",0
222. znakZaCijeliBrojBroj db "%d",0
223. znakZaNoviRedPlusNulZnak db 10,0
224. znakZaFloatPlusNoviRedPlusNulZnak db "%f",10,0
225. untrasnjaPetljaString db "Untrasnja petlja izvrsila se %d
puta.",10,0
226. slozenostString db "Ocekivani broj ponavljanja te petlje, po formuli
n*log2(n), bio bi %.1f.",10,0

```

```

227. sortiranjeJeTrajalo db "Sortiranje je trajalo %d milisekundi.",10,0
228. stringORazvrstanosti db "Razvrstanost pocetnog niza (s) iznosila je:
%f",10,0
229. izvjestajOFormuli db "Ocekivani broj usporedbi, po formuli: ",10
230. db "exp((ln(n)+ln(ln(n)))*1.05+(ln(n)-
ln(ln(n)))*0.83*abs(2.38854*pow(s,7)-0.284258*pow(s,6)-
1.87104*pow(s,5)+0.372637*pow(s,4)+0.167242*pow(s,3)-
0.0884977*pow(s,2)+0.315119*s))",10
231. db "bio bi: exp(%f)=%f",10,0
232. izvjestajOPolinomu:
233. if debug=1
234.     db "Polinom pod apsolutnom vrijednosti iznosi: %f",10
235. end if
236.     db 0
237.
238. section '.rdata' readable writable ; Deklaracije varijabli.
239. original dd 32768*4 DUP(?)
240. n dd ?

```

Kako compiler za AEC ne radi pretpostavke o tome kako je izvršna datoteka formatirana, ako ćete pisati program na jeziku AEC, morate sami deklarirati varijable na asemblerskom.

```
241. result dd ?
```

Compiler za AEC interno koristi varijablu zvanu `result`, nju je potrebno deklarirati, ali nije ju uputno koristiti.

```

242. brojac dd ?
243. pokazivac dd ?
244. i dd ?
245. stogSDonjimGranicama dd 32768*4 DUP(?)
246. stogSGornjimGranicama dd 32768*4 DUP(?)
247. pomocni dd 32768*4 DUP(?)
248. vrhStoga dd ?
249. donjaGranica dd ?
250. gornjaGranica dd ?
251. staviVece dd ?
252. staviManje dd ?
253. gdjeJePivot dd ?
254. procesorskoVrijeme dd ?
255. razvrstanost dd ?
256. razvrstanostNa dd 8 DUP(?)
257. polinomPodApsolutnom dd ?
258. eNaKoju dd ?
259. ocekivaniBrojUsporedbi dd ?
260. najmanjiCijeliBrojKojiSeMozeDodatiNaBrojac dd ?
261. pomocniBrojac dd ?
262. testZaPreljev dd ?
263.
264. section '.idata' data readable import ;Uvoz funkcija iz DLL-ova.

```

DLL znači *dinamička linkerska biblioteka*. Linker je, tako reći, program koji je rječnik manje poznatih riječi iz programskog jezika, kako compiler ne bi morao sve znati. DLL je skupina potprograma koji nisu povezani u program koji se može pokrenuti, nego ih se može pozvati iz drugih programa. I svaki današnji operativni sustav programima koji se na njemu vrte pruža linker koji omogućuje da se pozivaju potprogrami iz DLL-ova. To se zove *dinamičko linkiranje*. Linkeri koji se dobivaju uz compilere za C i slične jezike obično još omogućavaju i *statičko linkiranje*, da

se potprogrami iz DLL-ova ugrade u izvršni program, pa da on onda ne ovisi o DLL-ovima da bi se mogao vrtjeti. Linuxove dinamičke linkerske biblioteke imaju nastavak *.so* (*shared object*) umjesto *.dll*.

Treba razlikovati pojmove *dinamičko linkiranje* i *dinamičko programiranje*. Dinamičko programiranje je tehnika programiranja koja čini da se program vrti brže, ali da troši više memorije. To je gotovo suprotno od onog što radi dinamičko linkiranje. Dinamički programirani program može biti i dinamički i statički linkiran. Domagoj Kusalić u knjizi *Napredno programiranje i algoritmi u C-u i C++-u* tvrdi da je poznavanje tehnike dinamičkog programiranja korisno pri programiranju igara, no moje iskustvo to ne potvrđuje⁸.

265. `library msvcrt, 'msvcrt.dll' ; "msvcrt.dll" je stara verzija Microsoft Visual C Runtime Libraryja dostupna u C:\Windows\System32\msvcrt.dll na Windows 95 i novijim.`

Runtime library je DLL koji kao potprograme sadrži naredbe nekog jezika, najčešće je dio nekog kompilera.

Microsoft Visual C je Microsoftov compiler za programski jezik C. Da biste ga koristili, potrebna vam je licenca koju netko mora platiti Microsoftu. No, starija verzija *msvcrt.dll*, koji sadrži naredbe iz C-a kao potprograme, dobiva se uz Windowse, tako da ne morate imati Microsoft Visual C da biste mogli koristiti neku njegovu funkcionalnost, i da biste mogli koristiti programe koji su njime kompilirani. Statičko linkiranje s *runtime libraries*ima komercijalnih kompilera, i nekih besplatnih kompilera (ovisi o uvjetima korištenja), u mnogim je državama protuzakonito, kao i korištenje programa koji su tako linkirani. Čak i ako nije protuzakonito, to nije dobra ideja jer, kao prvo, programi koji su tako linkirani zauzimaju mnogo više prostora na tvrdom disku, kao drugo, teško je ili nemoguće provjeriti je li takav program stvarno linkiran s bibliotekom tog kompilera ili s nekom modificiranom verzijom koja sadrži virus.

266. `import msvcrt, printf, 'printf', system, 'system', exit, 'exit', scanf, 'scanf', clock, 'clock',`
`267. AsmEnd`

I, evo, kao što vidite, AEC-ov program i počinje i završava umetnutim asemblerskim kodom. To je posve druga paradigma nego kako se C odnosi prema asemblerskom jeziku, programi pisani u C-u i počinju i završavaju u C-u, a mogu sadržavati asemblerski kod jedino u sredini.

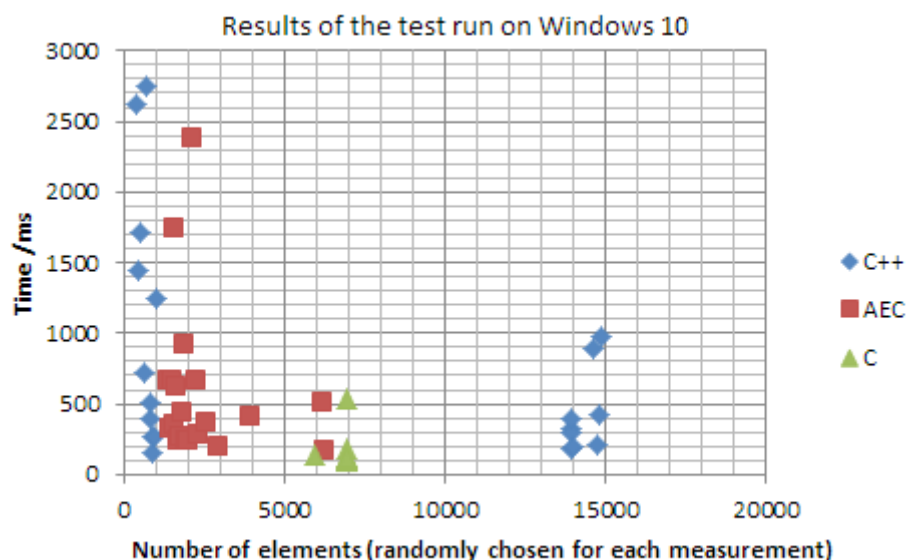
Testiranje i usporedba raznih implementacija brzog razvrstavanja

Na GitHubu je dostupan program koji će testirati daje li taj program u mom programskom jeziku isti rezultat kao i C++-ova naredba `sort` za velike nasumično generirane nizove, zove se *tester.cpp*⁹. Također je dostupan i C-ovski program koji je, prema autoru ovog teksta, ekvivalentan upravo opisanom AEC-ovskom programu¹⁰ (naravno, teško je odrediti znače li dvije rečenice na dvama različitim jezicima upravo isto, a isto ponekad vrijedi i za programske jezike, pogotovo kada govorimo o jezicima koji su toliko različiti kao što su C i AEC). Naravno, za očekivati je da je C++-ov `sort` najbrži, to je potprogram koji su pisali mnogo veći stručnjaci za informatiku nego što je autor ovog teksta. Također vjerojatno treba očekivati da je program pisan na C-u, kad se kompilira kvalitetnim C compilerima kao što je MinGW (vjerojatno najbolji besplatni compiler za C i C++ dostupan za Windows), brži od ekvivalentnog AEC-ovskog programa koji se kompilira AEC-ovim compilerom koji ni ne pokušava raditi optimizacije. No, kako to testirati? Ako bismo opet i opet pokrenuli *tester.cpp* i bilježili njegove rezultate, dobili bismo ovakav rezultat:

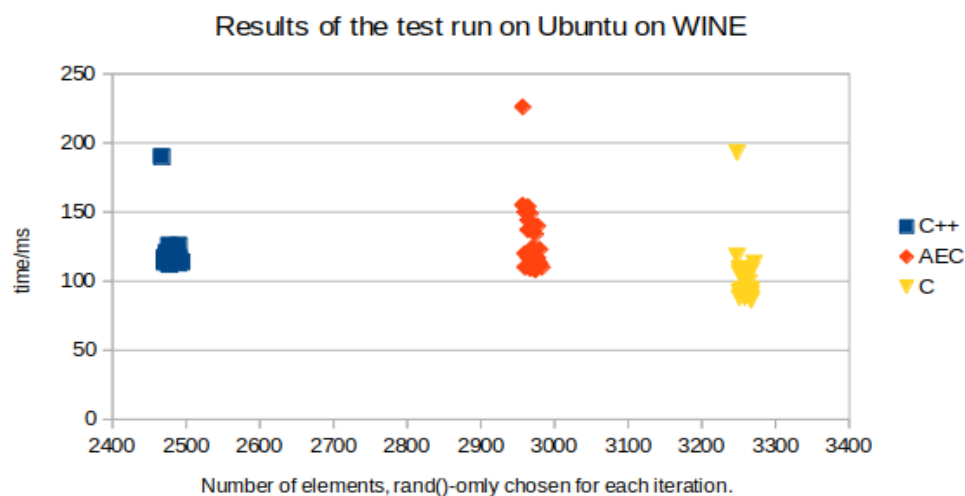
8 Programirao sam nekoliko jednostavnih igrica, recimo PacMan koji se može igrati na smartphoneima i flashcard igricu o povijesnoj lingvistici, i mogu komentirati da tehniku dinamičkog programiranja nisam mogao tamo nigdje upotrijebiti. Dostupne su na <https://flatassembler.000webhostapp.com/pacman.php> i <https://flatassembler.000webhostapp.com/etymologist.html> te na mom GitHub profilu.

9 <https://github.com/FlatAssembler/ArithmeticExpressionCompiler/blob/master/QuickSort/tester.cpp>

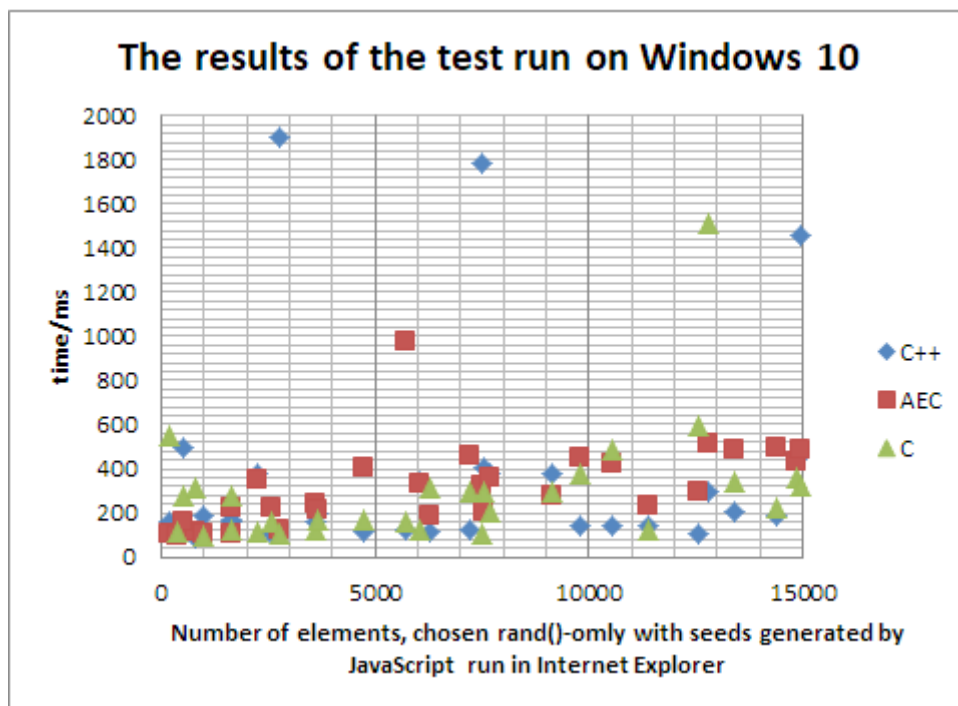
10 <https://github.com/FlatAssembler/ArithmeticExpressionCompiler/blob/master/QuickSort/qsort.c>



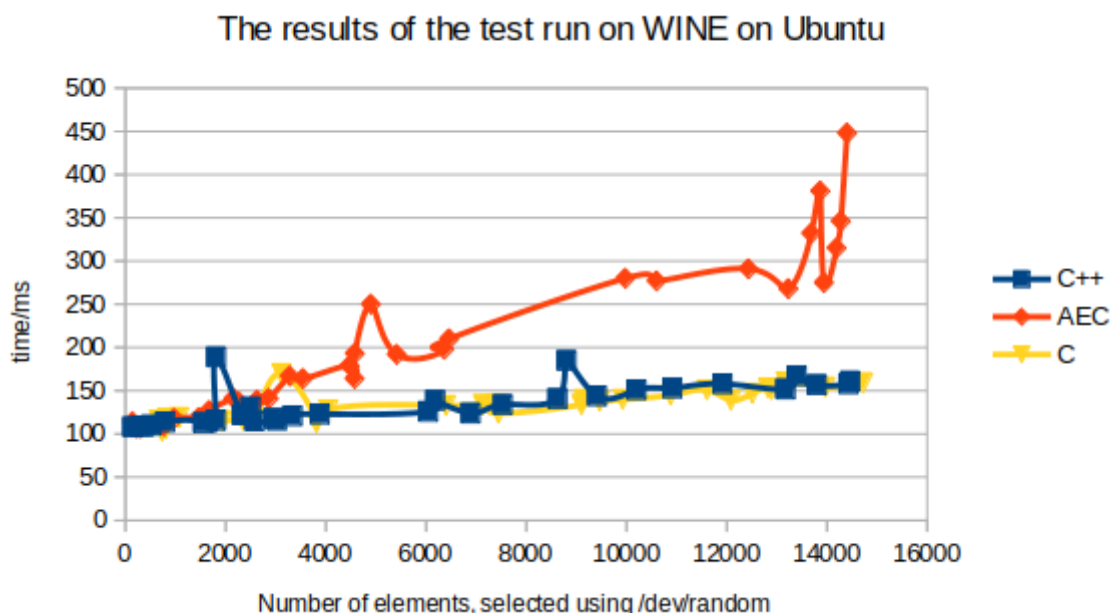
Naredba `rand` iz MSVCRT-a, za dobivanje nasumičnih brojeva, ne daje uniformne rezultate ako joj se zada da generira brojeve raspona od 0 do 15000. Dobro, sada, opće je poznato da se Windowsovi programi koji se mogu vrtjeti u WINE-u (besplatni program koji omogućuje da se neki Windowsovi programi vrte na Linuxu tako što presreće njihove pokušaje da pozovu potprograme iz Windowsovih sistemskih DLL-ova i preusmjerava te pozive na Linuxove SO-ove) općenito bolje vrte u WINE-u nego na Windowsima. Pa, probajmo onda to obaviti na Linuxu:



Ovo je zapravo još gore. Da, ti se programi vrte nekoliko puta brže na Linuxu nego na Windowsima na istom računalu, ali to znači da se C-ova naredba `srand` poziva opet i opet s istim sistemskim vremenom kao argumentom. Generirajmo sada na Windowsima mnogo nasumičnih brojeva koristeći JavaScript u Internet Exploreru, te ćemo onda njih davati kao argumente C-ovom `srand`-u:



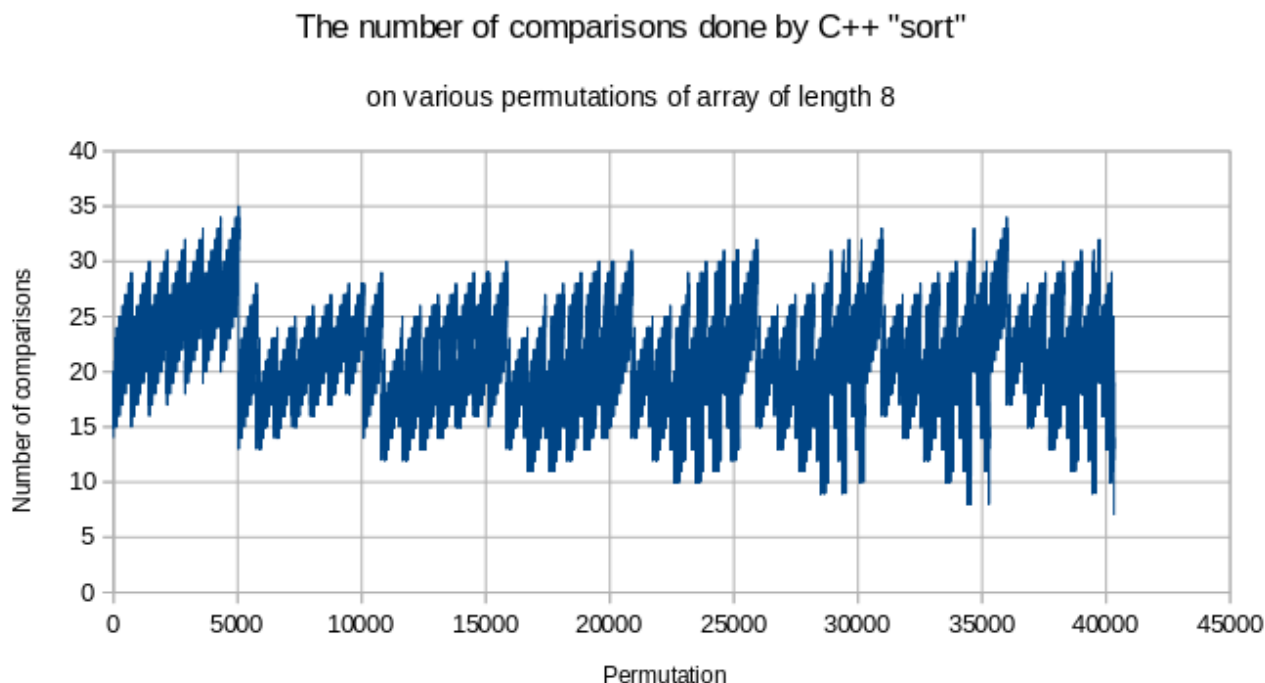
Ovdje se sada vidi da vrijeme izvršavanja programa za razvrstavanje raste s brojem elemenata koji mu se daju za razvrstavanje, no nije jasno koji je brži. To je zato što Windowsov scheduler (dio kernela koji detektira kad se neki program zaglavi i pokušava spriječiti da se zbog tog zaglave i drugi programi), kako je poznato, penalizira konzolne aplikacije, to jest, nasumično ih prekida u radu kako bi se Windowsovi sistemski procesi mogli brže izvršiti, i taj prekid, kako se čini, zna trajati i po dvije sekunde, mnogo više no što bilo kojem od tih programa treba za razvrstavanje niza. Napravimo sada sličan test na Linuxu, Linuxov scheduler to ne može raditi jer on ni ne zna je li program koji se vrti u WINE-u konzolna aplikacija:



Za one koji ne znaju, /dev/random Linuxov je driver za hardversku entropiju. Kada ga se pozove (to se na Linuxu radi tako da se jednostavno otvori datoteka /dev/random), on bi trebao davati binarne brojeve koji su onoliko nasumični koliko to hardver dopušta. Rezultati ovog eksperimenta već se mogu donekle protumačiti. Iz grafa se može očitati da za pokretanje programa

na WINE-u na mom laptopu (Acer Nitro 5) treba nešto malo više od 100 milisekundi. Također se vidi da je asemblerski kod koji generira MinGW približno tri puta brži od onoga koji generira moj compiler za AEC. Razlika između toga koliko se brzo vrti C++-ova naredba `sort` i toga koliko se brzo vrti moja implementacija QuickSort algoritma u C-u ovdje nije uočljiva.

Kako uopće funkcionira C++-ov `sort`? Koje on nizove razvrstava brzo, a koje sporo? Razvrstava li on brže nizove koji su već približno razvrstani, ili nizove koji još uopće nisu razvrstani. Znanstvena nulta hipoteza bila bi da on razvrstava jednako brzo nizove koji su već većinom razvrstani i nizove koji su potpuno nasumično poredani. To je značajka algoritma razvrstavanja spajanjem¹¹ (engleski *merge sort*), i zato se on koristi u sustavima stvarnog vremena, gdje je važna predvidljivost. Razvrstavanje spajanjem u većini je slučajeva sporije od QuickSorta (razvrstavanje spajanjem bazira se na činjenici da je spajanje dva poredana niza u jedan poredani niz moguće napraviti u linearnom vremenu, no to je, iako je još uvijek linearno vrijeme, na većini arhitektura računala znatno sporije nego premještanje niza kakvo radi QuickSort), ali je zato predvidljivo koliko će vremena trebati za razvrstavanje niza s određenim brojem elemenata: uvijek proporcionalno $n \cdot \log_2(n)$. Kako ćemo tu hipotezu testirati? Pa, C++ ima dobro poznatu naredbu za brzo traženje iduće permutacije nekog niza, ona se zove `next_permutation`¹². Naredba `sort` ima opcionalni argument kojim joj se može zadati da se neka naša funkcija koristi za uspoređivanje elemenata (umjesto C++-ovog operatora *manje od*, '<'). Pa, možemo u toj funkciji povećavati neki globalni brojač, da vidimo koliko je usporedbi za neku permutaciju niza napravila C++-ova naredba `sort`. Ako ona radi razvrstavanje spajanjem, broj usporedbi koje ona radi neće ovisiti o permutaciji niza, graf će biti horizontalna ravna crta. Ovo su rezultati tog testa:



Niz od 8 elemenata ima $8! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot 8 = 40320$ permutacija. Iz ovoga je očito da C++-ov `sort` ne koristi razvrstavanje spajanjem, najviše mu usporedbi treba da razvrsta približno 5000. permutaciju, tamo mu treba 35 usporedbi, a za 40320. permutaciju treba mu 7 usporedbi.

11 Implementaciju MergeSorta u AEC-u, s komentarima na hrvatskom, imate ovdje:

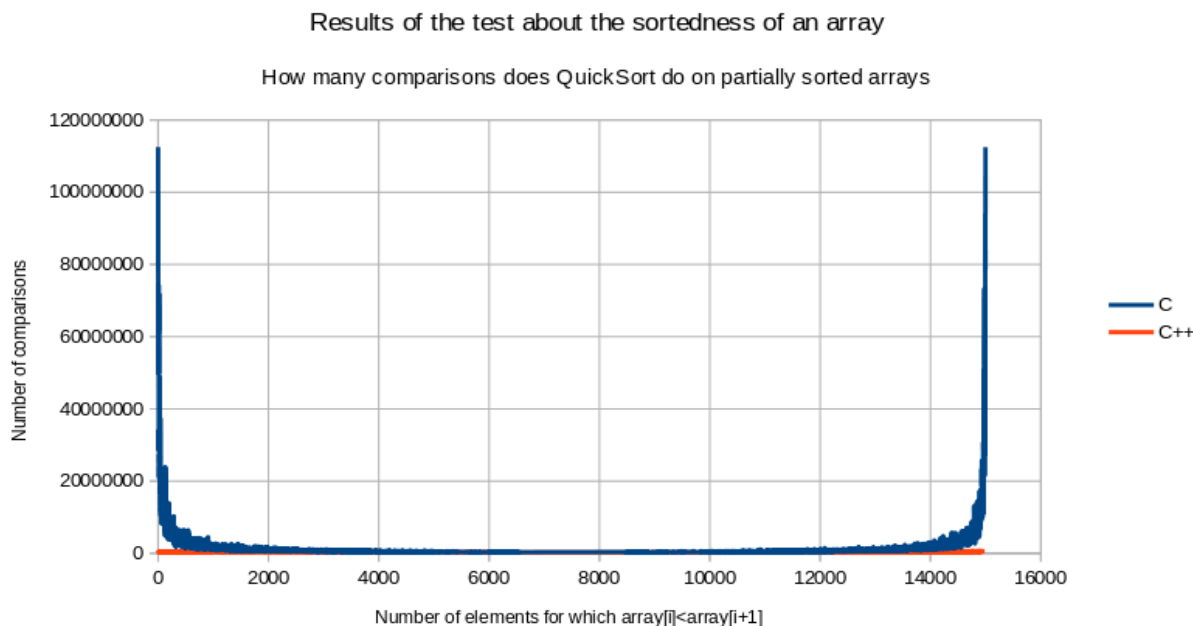
<https://github.com/FlatAssembler/ArithmeticExpressionCompiler/blob/master/MergeSort/msort.aec>

32-bitnu Windowsovu izvršnu datoteku imate ovdje:

<https://github.com/FlatAssembler/ArithmeticExpressionCompiler/raw/master/MergeSort/msort.exe>

12 Jedan od prvih programa koje sam napisao na AEC-u je implementacija algoritma permutacija. Izvorni kod i izvršna datoteka dostupni su na GitHubu: <https://github.com/FlatAssembler/ArithmeticExpressionCompiler/raw/master/ArithmeticExpressionCompiler.zip>

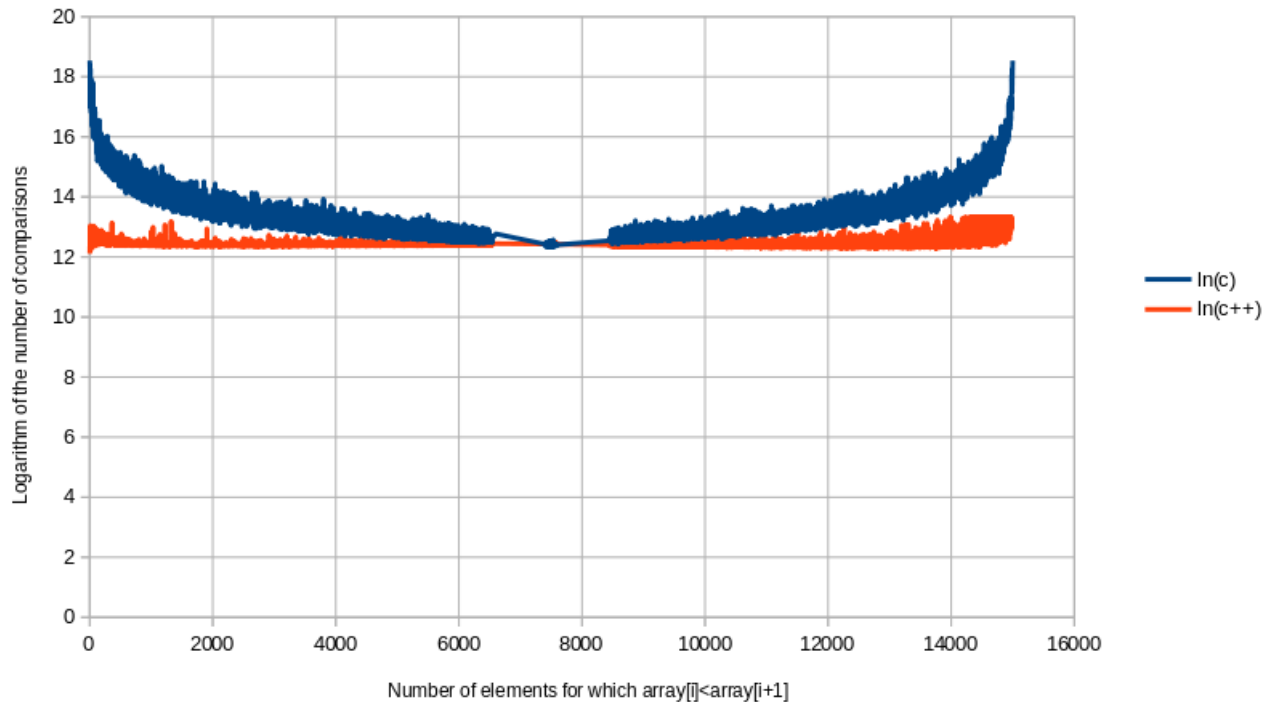
Kako ćemo to napraviti za veće nizove? Nema tog računala koje bi isprobalo sve permutacije nizova od nekoliko desetaka elemenata, a kamoli nekoliko stotina ili tisuća elemenata. Definirajmo za to pojam razvrstanost kao broj elemenata niza za koji vrijedi da je sljedeći element niza veći od njega, podijeljeno s polovicom broja elemenata u nizu, minus jedan. Kako nisam mogao naći englesku riječ za razvrstanost, odlučio sam izmisliti i englesku riječ za to: *sortedness*. Dakle, obrnuto poredani niz (od najvećeg prema najmanjem elementu) ima razvrstanost -1, nasumično poredani niz ima razvrstanost približno 0, a poredani niz ima razvrstanost 1. Razvrstanost nasumično izmiješanog niza moguće je odrediti u linearnom vremenu. Zato ćemo izmjeriti koliko će mom programu i C++-ovoj naredbi `sort` trebati usporedbi da razvrstaju nizove iste veličine, ali različite razvrstanosti:



Ovdje se jasno vidi jedan od paradoksa iz informatike: naivno implementirani QuickSort potrošit će najviše vremena pokušavajući poredati već poredani niz ili obrnuto poredani niz (kad je sve što treba napraviti za obrnuto poredani niz obrnuti ga u linearnom vremenu). Računalo ne ulazi u to ima li to što mu govori program da radi smisla ili nema. Njemu program kaže da opet i opet rastavi niz na niz s elementima manjim od prvog elementa i niz s elementima koji su veći od prvog elementa. Kako za već razvrstani niz nema elemenata koji su manji od prvog elementa, niz s elementima koji su veći od prvog elementa imat će samo jedan element manje od originalnog niza, pa će računalo onoliko puta koliko niz ima elemenata raditi to linearno rastavljanje niza na dva niza, i to će na kraju raditi u kvadratnom vremenu. I analogno vrijedi za obrnuto poredani niz. Za poredani niz od 15,000 elemenata, računalo je napravilo nešto manje od 120 milijuna usporedbi, dok je zapravo bilo dovoljno napraviti 14,999 usporedbi da se zaključi da je niz već poredan i da ne treba ništa raditi. Na ovakvom grafikonu vidi se jedino to da moj program troši hrpu vremena na razvrstavanje već poredanog ili obrnuto poredanog niza, ne vidi se uopće što se događa u sredini niti kako se ponaša C++-ov `sort`. Hajdemo ga prikazati u logaritamskoj skali, da velike promjene ne smetaju da se vide manje:

Results of the test about the sortedness of an array - logarithmic scale

How many comparisons does QuickSort do on partially sorted arrays?

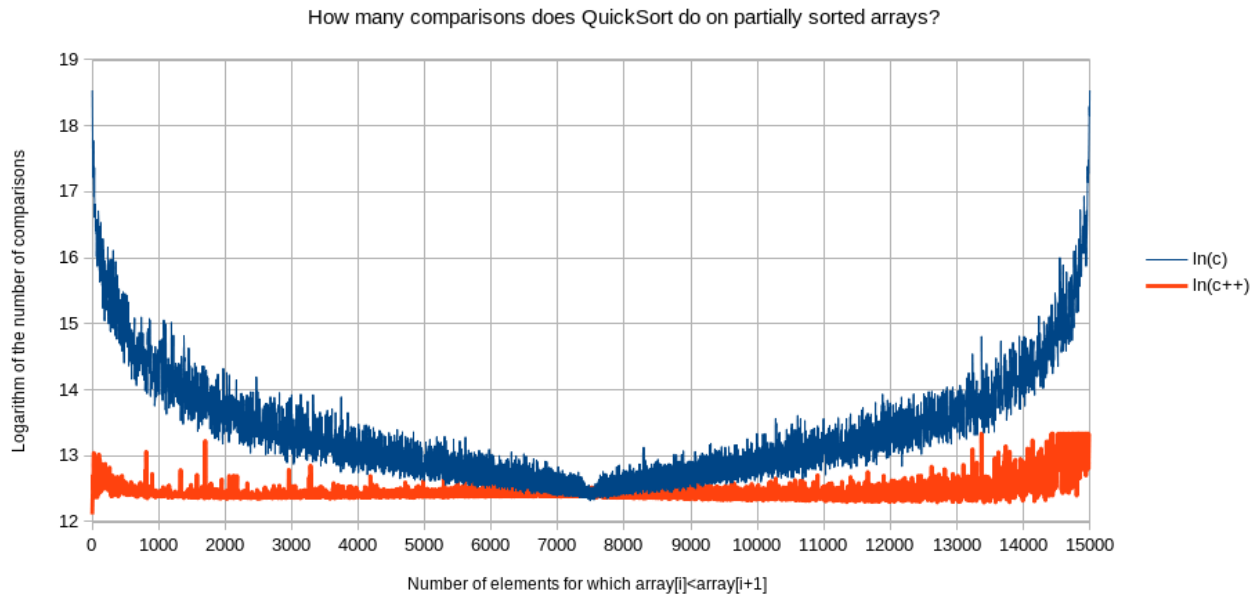


Dakle, C++-ov `sort` također počne biti nestabilan što se tiče broja usporedbi koje radi kako se razvrstanost udaljava od nule, no ne ni približno toliko koliko naivno implementirani QuickSort. Kada je razvrstanost približno 0 (kada je približno 7500 od 15000 poredano), naivna implementacija QuickSorta je u nekim slučajevima i brža od C++-ovog `sorta` (primijetite onu crvenu točkicu u gornjem desnom uglu mrljice na sredini krivulje), dok je inače znatno sporija. Ipak, ako bismo prije razvrstavanja nasumično ispremještali sve elemente u nizu, kao što radi C++-ova naredba `random_shuffle` u linearnom vremenu, razvrstanost bi uvijek bila veoma blizu nuli. Naime, ona mrljica u sredini krivulje predstavlja razvrstanosti kakve je dala ta C++-ova naredba.

Je li moguće u linearnom vremenu predvidjeti koliko će usporedbi QuickSort napraviti

Ovaj je seminar trebao biti prvenstveno o pitanju je li moguće u linearnom vremenu predvidjeti koliko će usporedbi QuickSort napraviti, kao što je za MergeSort moguće u konstantnom vremenu (ukoliko je broj elemenata u nizu unaprijed poznat, inače ih, naravno, moramo prebrojati u linearnom vremenu). Kako točno izgleda ta krivulja u logaritamskoj skali kad je razvrstanost osrednje udaljena od nule? Nasumično premještanje do $n/2$ nasumično odabranih elemenata u razvrstanom ili obrnuto razvrstanom nizu, kakvo je korišteno u gornja dva dijagrama, takve nam razvrstanosti ne daje. Pa hajdemo koristiti nasumično premještanje do n elemenata:

Results of the test about the sortedness of an array - logarithmic scale - array is shuffled two times



Kako bismo mogli matematički opisati tamnoplavu krivulju? Dakle, ona u nuli (na grafikonu je to 7500) ima šiljak prema dolje. Dakle, prva derivacija u nuli je beskonačna. Za koje je matematičke krivulje derivacija u nuli beskonačna? Pretpostavimo da se radi o apsolutnoj vrijednosti polinoma neparnog stupnja ovisnog o razvrstanosti, s korijenom u nula. To jest, pretpostavimo da bi funkcija dvije varijable, broj elemenata u polju i razvrstanost polja, koja kao rezultat daje broj usporedbi koje će QuickSort napraviti, imala ovakav oblik:

$$f(n, s) = e^{(\ln(n) + \ln(\ln(n))) \cdot C_1 + (\ln(n) - \ln(\ln(n))) \cdot C_2 \cdot |\text{neki} - \text{polinom} - \text{neparnog} - \text{stupnja}|}$$

Gdje je C_1 neki broj malo veći od 1, a C_2 neki broj malo manji od 1. Naime, QuickSort, kao što znaju već i oni koji imaju samo najosnovnije znanje iz informatike, u najboljem slučaju napravi približno $n \cdot \log_2(n)$ usporedbi. Kako taj izraz stavljamo u eksponent, moramo ga preoblikovati u $\ln(n \cdot \log_2(n)) = \ln(n) + \ln(\ln(n) \cdot C_1) \approx (\ln(n) + \ln(\ln(n))) \cdot C_1$, a, kako je binarni logaritam veći od prirodnoga, C_1 mora biti nešto veći od 1. Zapravo, kada ovako gledamo, trebalo bi biti $C_1 = 1/\ln(2) \approx 1.44$, no možda će računalo naći broj koji će bolje pristajati, jer evidentno je da QuickSort nekada napravi i nešto manje od $n \cdot \log_2(n)$ usporedbi. Malo je manje poznato da QuickSort u najgorem slučaju napravi nešto manje od n^2 usporedbi. Dakle, broj usporedbi koje

QuickSort napravi varira za faktor $\frac{n^2}{n \cdot \log_2(n)} = \frac{n}{\log_2(n)}$. Budući da taj izraz stavljamo u eksponent, preoblikujemo ga u:

$$\ln\left(\frac{n}{\log_2(n)}\right) = \ln(n) - \ln(\log_2(n)) = \ln(n) - \ln(\ln(n) \cdot C_2) \approx (\ln(n) - \ln(\ln(n))) \cdot C_2$$

Kako QuickSort uvijek napravi strogo manje od n^2 usporedbi, znamo da je C_2 manji od 1, no teško je reći koliko. No, razumno je pretpostaviti da je između 0.5 i 1.

Kombinirajući tehnike mekanog programiranja i objektivno orijentiranog programiranja napisao sam program koji traži taj polinom te brojeve C_1 i C_2 , pod pretpostavkom da je formula uistinu takvog oblika, dostupan je na GitHubu¹³. Naletio sam na naizgled neobjašnjiv problem pri kompiliranju, naime, kada sam compileru GCC 8.3 zadao da kompilira taj program, linker se srušio s nekom nerazumljivom porukom o pogrešci. Uspio sam zaobići taj problem tako što sam mu zadao da linkira statički umjesto dinamički. Isprva sam mislio da sam pogriješio u nečemu vezanom za

13 https://github.com/FlatAssembler/ArithmeticExpressionCompiler/blob/master/QuickSort/Genetic_algorithm_for_deriving_the_formula/genetic.cpp

objektivno orijentirano programiranje, u čemu nisam pretjerano vješt. Da stvar bude čudnija, CLANG 9.0 je mogao dati 32-bitnu Linuxovu dinamički linkiranu izvršnu datoteku, a MinGW 8.2 (koji većinom dijeli isti izvorni kod kao GCC 8.3) na Windows 10 nije pravio nikakve probleme. Kako netko tko želi pokrenuti taj C++-ov program ne bi naletio na teško razumljive probleme s kompiliranjem, objavio sam 64-bitnu Linuxovu, 32-bitnu Linuxovu i 32-bitno Windowsovu izvršnu datoteku na GitHub¹⁴. Kasnije sam shvatio da nije riječ o objektivno orijentiranom programiranju, niti toliko o compilerima, koliko o Linuxu. 64-bitni programi za Linux većinom očekuju (razumno) da su Linuxove systemske datoteke `/usr/local/lib64/libstdc++.so.6` i `/usr/lib64/libstdc++.so.6` identične. To na mom laptopu nije bio slučaj. Ja sam iz izvornog koda compilirao novu verziju `libstdc++-a` i spremio sam je u `/usr/local/lib64`, a u `/usr/lib64` ostala je starija i ne posve kompatibilna verzija. Problem s time da ne mogu dobiti 64-bitne dinamički linkirane izvršne datoteke za kompliciranije C++ programe (za *Hello World* program sam mogao) nestao je kada sam kopirao `/usr/local/lib64/libstdc++.so.6` u direktorij `/usr/lib64` i zamijenio postojeću datoteku.

Dakle, taj program je, nakon što se nekoliko minuta vrtio na mom laptopu, ispisao da smatra da je rješenje:

$$f(n, s) = \exp((\ln(n) + \ln(\ln(n))) * 1.05 + (\ln(n) - \ln(\ln(n))) * 0.83 * \text{abs}(2.38854 * \text{pow}(s, 7) - 0.284258 * \text{pow}(s, 6) - 1.87104 * \text{pow}(s, 5) + 0.372637 * \text{pow}(s, 4) + 0.167242 * \text{pow}(s, 3) - 0.0884977 * \text{pow}(s, 2) + 0.315119 * s))$$

Drugim riječima, da vrijedi:

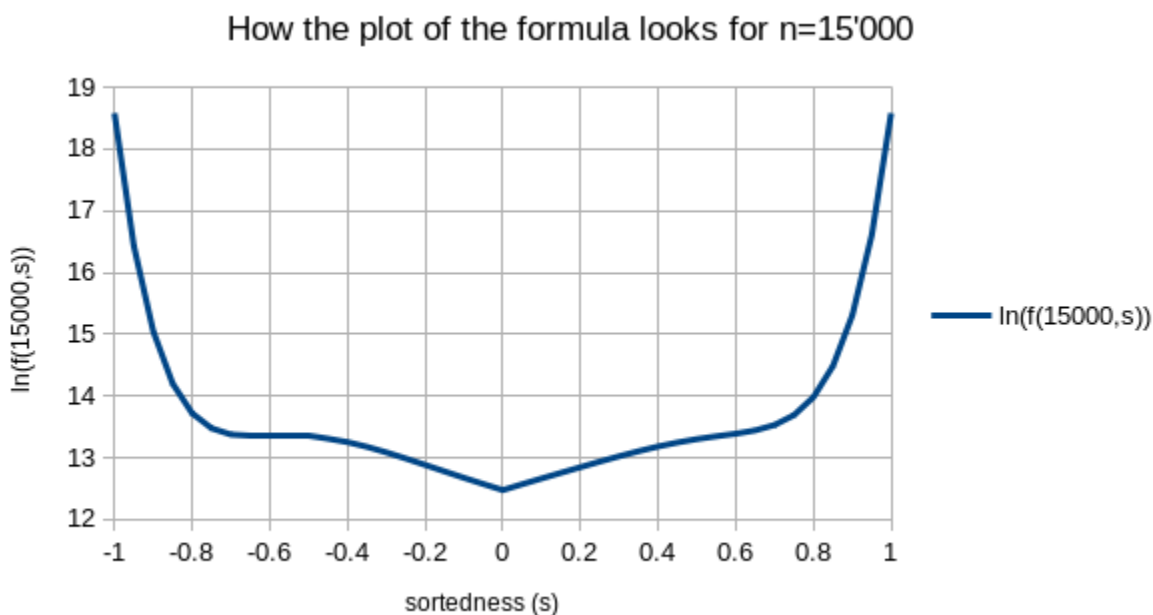
$C_1 \approx 1.05$

$C_2 \approx 0.83$

Te da onaj polinom iznosi približno:

$$2.38854 \cdot s^7 - 0.284258 \cdot s^6 - 1.87104 \cdot s^5 + 0.372637 \cdot s^4 + 0.167242 \cdot s^3 - 0.0884977 \cdot s^2 + 0.315119 \cdot s$$

Nadam se da se slažete da je znanstveno-metodološki ispravna hipoteza da je ta formula ispravna. Evo kako ta formula izgleda za $n=15000$, usporedite je s plavom krivuljom na prethodnom dijagramu:



¹⁴ Linkovi su ovdje: https://github.com/FlatAssembler/ArithmeticExpressionCompiler/tree/master/QuickSort/Genetic_algorithm_for_deriving_the_formula

Genetski algoritam iz onog objektivno-orijentiranog i mekano programiranog programa imao je pristup podacima za $n=15000$, pa je po njima namještao brojeve iz formule. Znanstveno testiranje hipoteze da je ta formula ispravna bilo bi da je isprobamo na n -ovima koji su znatno različiti od toga. Zato sam je ugradio u onaj program za razvrstavanje u svom programskom jeziku, da bude lakše isprobati je za manje nizove.

Za već poredani niz veličine $n=10$, ta formula predviđa da će QuickSortu trebati 91 usporedba da razvrsta niz. Zapravo mu, mjerenje pokazuje, treba 45 usporedbi. To je greška od $(91-45)/45=102\%$. Za poredani niz veličine $n=100$, ta formula predviđa da će QuickSortu trebati 8047 usporedbi. Mjerenje pokazuje da mu treba 4950 usporedbi. To je greška od $(8047-4950)/4950=62\%$. Formula predviđa da će QuickSort za poredani niz od 1000 elemenata napraviti 666102 usporedbe. Mjerenje pokazuje da on za takav niz radi 499500 usporedbi. To je greška od $(666102-499500)/499500=33\%$. Mislim da je iz ovoga jasno da je ta formula samo prividno točna za $n=15000$, i da za manje nizove ona drastično precijeni koliko će QuickSort napraviti usporedbi.

Bi li ta formula bila korisna za hibridni algoritam sastavljen od QuickSorta i MergeSorta

Autoru ovog teksta učinilo se da bi takva formula, kad bi bila točna, bila korisna da se implementira hibridni algoritam koji se ponaša kao MergeSort kad je niz približno poredan po veličini, a kao QuickSort kad je niz nasumično poredan. Tu je hipotezu već sada moguće provjeriti eksperimentalno, jer ta formula ipak je približno točna ako je broj elemenata u nizu približno 15000. Zato je autor ovog teksta napisao u AEC-u hibridni algoritam koji po toj formuli odlučuje hoće li se ponašati kao QuickSort ili kao MergeSort¹⁵:

```
1. ;HybridSort algoritam - kombinacija QuickSort algoritma i MergeSort
   algoritma.
2. AsmStart
3.     ispisPoruka=1
4.     macro staviIntNaSistemiStog x ;"x" treba biti pokazivac na 32-bitni
       decimalni broj ("float"), kojeg ce ova makro-naredba pretvoriti u 32-bitni
       cijeli broj ("int") i staviti na sistemi stog.
5.     {
6.         sub esp,4
7.         fld dword [x]
8.         fistp dword [esp]
9.     }
10.    macro staviPokazivacNaSistemiStog x
11.    {
12.        sub esp,4
13.        lea ebx,[x]
14.        mov [esp],ebx
15.    }
16.    macro staviStringNaSistemiStog x
17.    {
18.        sub esp,4
19.        mov dword [esp],x
20.    }
21.    format PE console ;"PE" je 32-bitna Windowsova ".EXE" datoteka (to
       nije sve sto FlatAssembler moze stvarati).
22.    entry start
23.
```

15 Izvorni kod dostupan je i na: <https://github.com/FlatAssembler/ArithmeticExpressionCompiler/blob/master/HybridSort/hsort.aec>

32-bitna Windowsova izvršna datoteka dostupna je ovdje:

<https://github.com/FlatAssembler/ArithmeticExpressionCompiler/raw/master/HybridSort/hsort.exe>

```

24.    include 'win32a.inc' ;FlatAssemblerove naredbe za upravljanje DLL-
      ovima (ovdje se koriste za pozivanje C-ovih funkcija iz MSVCRT-a).
25.
26.    section '.text' code executable
27.    start:
28. if ispisPoruka=1
29.    jmp velicinaUnosa$
30.        velicinaUnosa db "Unesite koliko cete brojeva unijeti.",10,0
31.    velicinaUnosa$:
32.    staviStringNaSistemiStog velicinaUnosa
33.    call [printf]
34. end if
35.    staviPokazivacNaSistemiStog n
36.    jmp znakZaFloat$
37.        znakZaFloat db "%f",0
38.    znakZaFloat$:
39.    staviStringNaSistemiStog znakZaFloat
40.    call [scanf]
41.    if ispisPoruka=1
42.        jmp pitajZaUnos$
43.        pitajZaUnos db "Unesite te brojeve:",10,0
44.        pitajZaUnos$:
45.        staviStringNaSistemiStog pitajZaUnos
46.        call [printf]
47.    end if
48.    AsmEnd
49. i:=0
50. brojac:=0
51. vrhStoga:=0
52. While i<n
53.    pokazivac:=4*i ;"float" ima 4 bajta.
54.    AsmStart
55.        fld dword [pokazivac]
56.        fistp dword [pokazivac]
57.        lea ebx,[original]
58.        add ebx,[pokazivac]
59.        staviPokazivacNaSistemiStog ebx
60.        staviStringNaSistemiStog znakZaFloat
61.        call [scanf]
62.    AsmEnd
63.    i:=i+1
64.    EndWhile
65.    AsmStart
66.    call [clock] ;"clock" na Windowsima vraća broj milisekundi otkad se
      program pokrenuo, zadnja 32 bita vraća u procesorski registar "eax".
67.    mov [procesorskoVrijeme],eax
68.    AsmEnd
69. razvrstanost:=0
70. i:=0
71. While i<n-1
72.    razvrstanost:=razvrstanost+(original(i)<original(i+1))
73.    i:=i+1

```

```

74.     brojac:=brojac+1
75.     EndWhile
76. razvrstanost:=razvrstanost/((n-1)/2)-1
77.     AsmStart
78.     if ispisPoruka=1
79.         jmp izvjesceORazvrstanosti$
80.         izvjesceORazvrstanosti db "Razvrstanost pocetnog niza iznosi:
%f",10,0
81.         izvjesceORazvrstanosti$:
82.         fld dword [razvrstanost]
83.         fstp qword [esp]
84.         staviStringNaSistemskiStog izvjesceORazvrstanosti
85.         call [printf]
86.     end if
87.     AsmEnd
88. i:=2
89. While i<7 | i=7
90.     razvrstanostNa(i):=pow(abs(razvrstanost),i) ;"pow(x,y)" je u AEC-u
samo sintaksni secer za "exp(ln(x)*y)", i to vraca NaN za x=0 ili x<0.
Nema ocitog nacina da se "pow(x,y)" prevede na asemblerski.
91.     If razvrstanost=0
92.         razvrstanostNa(i):=0
93.     EndIf
94.     If mod(i,2)=1 & razvrstanost<0
95.         razvrstanostNa(i):=-razvrstanostNa(i)
96.     EndIf
97.     i:=i+1
98.     EndWhile
99.     ;Formula koju je ispisao genetski algoritam za predvidanje koliko ce
usporedbi QuickSort napraviti:
https://github.com/FlatAssembler/ArithmeticExpressionCompiler/tree/master/QuickSort/Genetic\_algorithm\_for\_deriving\_the\_formula
100.    polinomPodApsolutnom:=2.38854*razvrstanostNa(7)-
0.284258*razvrstanostNa(6)-
1.87104*razvrstanostNa(5)+0.372637*razvrstanostNa(4)+0.167242*razvrstanost
Na(3)-0.0884977*razvrstanostNa(2)+0.315119*razvrstanost
101.    eNaKoju:=(ln(n)+ln(ln(n)))*1.05+(ln(n)-
ln(ln(n)))*0.83*abs(polinomPodApsolutnom)
102.    kolikoUsporedbiOcekujemoOdQuickSorta:=exp(eNaKoju)
103.    kolikoUsporedbiOcekujemoOdMergeSorta:=2*n*ln(n)/ln(2)
104.    AsmStart
105.    if ispisPoruka=1
106.        jmp ispisOTomeStoOcekujemo$
107.        ispisOTomeStoOcekujemo db "Od QuickSorta ocekujemo %f
usporedbi, a od MergeSorta ocekujemo %f usporedbi.",10,0
108.        ispisOTomeStoOcekujemo$:
109.        fld dword [kolikoUsporedbiOcekujemoOdMergeSorta]
110.        fstp qword [esp+8]
111.        fld dword [kolikoUsporedbiOcekujemoOdQuickSorta]
112.        fstp qword [esp]
113.        staviStringNaSistemskiStog ispisOTomeStoOcekujemo
114.        call [printf]
115.    end if
116.    AsmEnd

```

```

117.     najmanjiCijeliBrojKojiSeMozeDodatiNaBrojac:=1 ;Da, kada prijedemo
MAX_SAFE_INTEGER za "float", ne pokušavamo više dodavati jedinicu.
118.     pomocniBrojac:=0
119.     If razvrstanost=1
120.         AsmStart
121.             if ispisPoruka=1
122.                 jmp nizJeVecRazvrstan$
123.                 nizJeVecRazvrstan db "Niz je vec poredan, ne radimo
nista.",10,0
124.                 nizJeVecRazvrstan$:
125.                 invoke printf,nizJeVecRazvrstan
126.             end if
127.         AsmEnd
128.     ElseIf razvrstanost=(-1) ;Da, parser od AEC-a jos nije savrsen i
javlja gresku za "razvrstanost=-1".
129.         AsmStart
130.             if ispisPoruka=1
131.                 jmp nizJeObrnutoRazvrstan$
132.                 nizJeObrnutoRazvrstan db "Niz je obrnuto
poredan.",10,0
133.                 nizJeObrnutoRazvrstan$:
134.                 invoke printf,nizJeObrnutoRazvrstan
135.             end if
136.         AsmEnd
137.         i:=0
138.         While i<n
139.             pomocni(i):=original(n-i-1)
140.             i:=i+1
141.             brojac:=brojac+1
142.         EndWhile
143.         i:=0
144.         While i<n
145.             original(i):=pomocni(i)
146.             i:=i+1
147.         EndWhile
148.     ElseIf
kolikoUsporedbiOcekujemoOdQuickSorta<kolikoUsporedbiOcekujemoOdMergeSorta
149.         AsmStart
150.             if ispisPoruka=1
151.                 jmp radimoQuickSort$
152.                 radimoQuickSort db "Primijenit cemo QuickSort
algoritam.",10,0
153.                 radimoQuickSort$:
154.                 invoke printf,radimoQuickSort
155.             end if
156.         AsmEnd
157.         vrhStoga:=vrhStoga+1
158.         stogSDonjimGranicama(vrhStoga):=0
159.         stogSGornjimGranicama(vrhStoga):=n
160.         While vrhStoga>0
161.             gornjaGranica:=stogSGornjimGranicama(vrhStoga)
162.             donjaGranica:=stogSDonjimGranicama(vrhStoga)

```

```

163.         vrhStoga:=vrhStoga-1
164.         gdjeJePivot:=donjaGranica
165.         i:=donjaGranica+1
166.         While i<gornjaGranica
167.             If original(i)<original(donjaGranica)
168.                 gdjeJePivot:=gdjeJePivot+1 ;Gdje ce doci element
koji je sada prvi ("pivot").
169.             EndIf
170.             i:=i++ ;"++" je u AEC-u jednostavno sintaksni secer za
"+1".
171.         EndWhile
172.         staviManje:=donjaGranica
173.         staviVece:=gdjeJePivot+1
174.         pomocni(gdjeJePivot):=original(donjaGranica)
175.         i:=donjaGranica+1
176.         While i<gornjaGranica ;Preuredi niz
original(donjaGranica..gornjaGranica-1) tako da svi elementi koji su manji
od onoga koji je bio prvi dodu prije njega.
177.             If original(i)<original(donjaGranica)
178.                 pomocni(staviManje):=original(i)
179.                 staviManje:=staviManje+1
180.             Else
181.                 pomocni(staviVece):=original(i)
182.                 staviVece:=staviVece+1
183.             EndIf
184.             pomocniBrojac:=pomocniBrojac+1
185.             If
pomocniBrojac=najmanjiCijeliBrojKojiSeMozeDodatiNaBrojac
186.                 brojac:=brojac+pomocniBrojac
187.                 pomocniBrojac:=0
188.             EndIf
189.             i:=i+1
190.         EndWhile
191.         i:=donjaGranica
192.         While i<gornjaGranica
193.             original(i):=pomocni(i)
194.             i:=i+1
195.         EndWhile
196.         ;Razdvoji niz original(donjaGranica..gornjaGranica-1) na
nizove original(donjaGranica..gdjeJePivot-1) i
original(gdjeJePivot+1..gornjaGranica-1).
197.         ;Znamo gdje je pivot, pa njega ne trebamo ukljuciti ni u
jedan od tih nizova.
198.         ;I ne trebamo na stog stavljati naputke o razvrstavanju
nizova velicine 0 ili 1.
199.         If gdjeJePivot<gornjaGranica-1
200.             vrhStoga:=vrhStoga+1
201.             stogSDonjimGranicama(vrhStoga):=gdjeJePivot+1
202.             stogSGornjimGranicama(vrhStoga):=gornjaGranica
203.         EndIf
204.         If gdjeJePivot>donjaGranica+1
205.             vrhStoga:=vrhStoga+1
206.             stogSDonjimGranicama(vrhStoga):=donjaGranica

```



```

207.          stogSGornjimGranicama (vrhStoga) :=gdjeJePivot
208.          EndIf
209.          testZaPreljev:=brojac+najmanjiCijeliBrojKojiSeMozeDodatiNaBrojac ;Potrebna
           je posebna varijabla za to jer FPU interno radi s 80-bitnim brojevima, a
           CPU s 32-bitnim. Izgubio sam hrpu vremena da to shvatim.
210.          If not (testZaPreljev>brojac)
211.          najmanjiCijeliBrojKojiSeMozeDodatiNaBrojac:=najmanjiCijeliBrojKojiSeMozeDo
           datiNaBrojac*2
212.          AsmStart
213.              if ispisPoruka=1
214.                  jmp izvjesceOpreljevu$
215.                  izvjesceOpreljevu db "Upozorenje: Brojac
           mozda nece sadrzavati tocan rezultat, dogodio se preljev na %d.
           iteraciji."
216.                  db " Najveca ocekivana pogreska za ovaj
           preljev je %d krivo prebrojanih izvrsavanja unutarnje petlje.",10,0
217.                  izvjesceOpreljevu$:
218.                  fld dword [gornjaGranica]
219.                  fld dword [donjaGranica]
220.                  fsubp
221.                  fabs
222.                  fistp dword [esp+4]
223.                  fld dword [brojac]
224.                  fistp dword [esp]
225.                  invoke printf,izvjesceOpreljevu
226.                  end if
227.          AsmEnd
228.          EndIf
229.          EndWhile
230.      Else
231.          AsmStart
232.              if ispisPoruka=1
233.                  jmp radimoMergeSort$
234.                  radimoMergeSort db "Primijenit cemo MergeSort
           algoritam.",10,0
235.                  radimoMergeSort$:
236.                  invoke printf,radimoMergeSort
237.                  end if
238.          AsmEnd
239.          vrhStoga:=vrhStoga+1
240.          stogSDonjimGranicama (vrhStoga) :=0
241.          stogSGornjimGranicama (vrhStoga) :=n
242.          stogSPodacimaTrebaLiPetljaRazdvajatiIliSpajatiNizove (vrhStoga) :=razdvajati
243.          While vrhStoga>0
244.              gornjaGranica:=stogSGornjimGranicama (vrhStoga)
245.              donjaGranica:=stogSDonjimGranicama (vrhStoga)
246.              trebaLiSpajatiIliRazdvajati:=stogSPodacimaTrebaLiPetljaRazdvajatiIliSpajat
           iNizove (vrhStoga)
247.              vrhStoga:=vrhStoga-1
248.              sredinaNiza:=(donjaGranica+gornjaGranica)/2

```

```

249.          sredinaNiza:=sredinaNiza-mod(sredinaNiza,1)
250.          If trebaLiSpajatiIliRazdvajati=razdvajati ;Razdvoji niz
            original(donjaGranica..gornjaGranica-1) na
            original(donjaGranica..sredinaNiza-1) i
            original(sredinaNiza..gornjaGranica-1) .
251.          If gornjaGranica-donjaGranica>1 ;Niz velicine 0 ili 1
            vec je poredan i ne radimo nista dalje.
252.          vrhStoga:=vrhStoga+1
253.          stogSDonjimGranicama(vrhStoga):=donjaGranica
254.          stogSGornjimGranicama(vrhStoga):=gornjaGranica
255.          stogSPodacimaTrebaLiPetljaRazdvajatiIliSpajatiNizove(vrhStoga):=spajati
256.          ;Stavljamo naputak za spajanje nizova prvog na stog
            kako bi on onda bio zadnji izvađen iz njega.
257.          vrhStoga:=vrhStoga+1
258.          stogSDonjimGranicama(vrhStoga):=donjaGranica
259.          stogSGornjimGranicama(vrhStoga):=sredinaNiza
260.          stogSPodacimaTrebaLiPetljaRazdvajatiIliSpajatiNizove(vrhStoga):=razdvajati
261.          vrhStoga:=vrhStoga+1
262.          stogSDonjimGranicama(vrhStoga):=sredinaNiza
263.          stogSGornjimGranicama(vrhStoga):=gornjaGranica
264.          stogSPodacimaTrebaLiPetljaRazdvajatiIliSpajatiNizove(vrhStoga):=razdvajati
265.          EndIf
266.          Else ;Spoji vec poredane nizove
            original(donjaGranica..sredinaNiza-1) i
            original(sredinaNiza..gornjaGranica-1) u novi poredani niz
            original(donjaGranica..gornjaGranica-1) .
267.          i:=donjaGranica
268.          gdjeSmoUPrvomNizu:=donjaGranica
269.          gdjeSmoUDrugomNizu:=sredinaNiza
270.          While i<gornjaGranica
271.              If (gdjeSmoUPrvomNizu=sredinaNiza |
                original(gdjeSmoUDrugomNizu)<original(gdjeSmoUPrvomNizu)) &
                gdjeSmoUDrugomNizu<gornjaGranica
272.                  pomocni(i):=original(gdjeSmoUDrugomNizu)
273.                  gdjeSmoUDrugomNizu:=gdjeSmoUDrugomNizu+1
274.              Else
275.                  pomocni(i):=original(gdjeSmoUPrvomNizu)
276.                  gdjeSmoUPrvomNizu:=gdjeSmoUPrvomNizu+1
277.              EndIf
278.              i:=i+1
279.              brojac:=brojac+1
280.          EndWhile
281.          i:=donjaGranica
282.          While i<gornjaGranica
283.              original(i):=pomocni(i)
284.              brojac:=brojac+1
285.              i:=i+1
286.          EndWhile
287.          EndIf
288.          EndWhile
289.          EndIf

```

```

290.      AsmStart
291.          call [clock]
292.          sub eax,[procesorskoVrijeme]
293.          mov [procesorskoVrijeme],eax
294.      if ispisPoruka=1
295.          jmp sortiraniNizJe$
296.          sortiraniNizJe db "Sortirani niz je:",10,0
297.          sortiraniNizJe$:
298.          staviStringNaSistemskeStog sortiraniNizJe
299.          call [printf]
300.      end if
301.      AsmEnd
302.      i:=0
303.      While i<n
304.          pokazivac:=4*i
305.          AsmStart
306.              lea ebx,[original]
307.              fld dword [pokazivac]
308.              fistp dword [pokazivac]
309.              add ebx,[pokazivac]
310.              fld dword [ebx]
311.              fstp qword [esp]
312.              staviStringNaSistemskeStog znakZaFloatPlusNoviRedPlusNulZnak
313.              call [printf]
314.          AsmEnd
315.          i:=i+1
316.      EndWhile
317.      AsmStart
318.      if ispisPoruka=1
319.          staviIntNaSistemskeStog broj
320.          staviStringNaSistemskeStog unutrašnjaPetljaString
321.          call [printf]
322.      AsmEnd
323.      n*ln(n)/ln(2) ;Ovo će se spremati u "result", pomoćnu varijablu
koju koristi compiler za AEC.
324.      AsmStart
325.          fld dword [result]
326.          fstp qword [esp] ;"printf" iz MSVCRT-a za "%f" očekuje 64-bitni
"double", ili, na asemblerskom jeziku, "qword".
327.          staviStringNaSistemskeStog složenostString
328.          call [printf]
329.          push dword [procesorskoVrijeme]
330.          invoke printf,sortiranjeJeTrajalo
331.          invoke system,_pause
332.      end if
333.      invoke exit,0
334.
335.      ;"Konstante", ako njih pokušamo mijenjati, dobijemo Segmentation
Fault:
336.      _pause db "PAUSE",0
337.      znakZaCijeliBrojBroj db "%d",0
338.      znakZaNoviRedPlusNulZnak db 10,0
339.      znakZaFloatPlusNoviRedPlusNulZnak db "%f",10,0

```

```

340.      untrasnjaPetljaString db "Untrasnja petlja izvorsila se %d
      puta.",10,0
341.      slozenostString db "Ocekivani broj ponavljanja te petlje, po formuli
      n*log2(n), bio bi %.1f.",10,0
342.      sortiranjeJeTrajalo db "Sortiranje je trajalo %d milisekundi.",10,0
343.      razdvajati dd 0f
344.      spajati dd 1f
345.
346.      section '.rdata' readable writable ;Varijable i polja (u drugom
      segmentu programa nego sto je izvrsni dio).
347.      original:
348.      repeat 32768 ;Nije preporucljivo ovako na asemblerskom
      deklarirati nizove, ali zasto bih se pretvarao da radim za racunalom s 4
      MB RAM-a, gdje je problem učitati program gdje je jedan segment velik 640
      KB?

```

Naime, autor je bio naletio na njemu neobjašnjiv problem da mu se program ruši ako mu se zada niz veći od približno 17000 elemenata. Pokušao ga je riješiti ovako, no ni to nije uspješno.

```

349.      dd 0
350.      end repeat
351.      n dd ?
352.      result dd ?
353.      brojac dd ?
354.      pokazivac dd ?
355.      i dd ?
356.      stogSDonjimGranicama:
357.      repeat 32768
358.      dd 0
359.      end repeat
360.      stogSGornjimGranicama:
361.      repeat 32768
362.      dd 0
363.      end repeat
364.      pomocni:
365.      repeat 32768
366.      dd 0
367.      end repeat
368.      vrhStoga dd ?
369.      donjaGranica dd ?
370.      gornjaGranica dd ?
371.      staviVece dd ?
372.      staviManje dd ?
373.      gdjeJePivot dd ?
374.      procesorskoVrijeme dd ?
375.      razvrstanost dd ?
376.      razvrstanostNa dd 8 DUP(?)
377.      polinomPodApsolutnom dd ?
378.      eNaKoku dd ?
379.      kolikoUsporedbiOcekujemoOdQuickSorta dd ?
380.      kolikoUsporedbiOcekujemoOdMergeSorta dd ?
381.      najmanjiCijeliBrojKojiSeMozeDodatiNaBrojac dd ?
382.      pomocniBrojac dd ?
383.      testZaPreljev dd ?

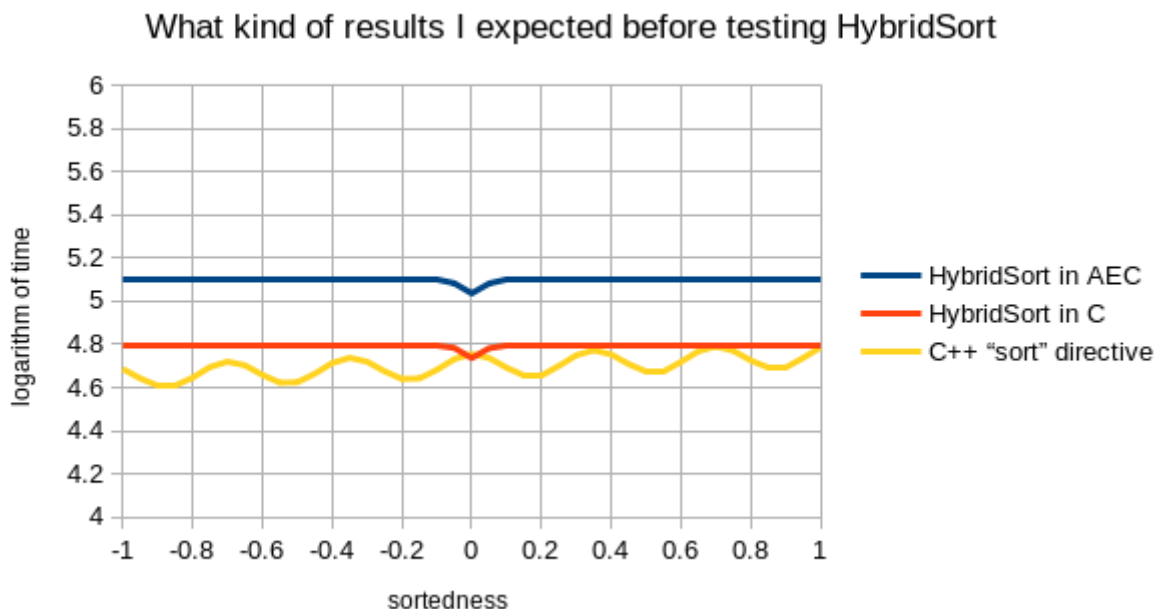
```

```

384.         gdjeSmoUDrugomNizu dd ?
385.         gdjeSmoUPrvomNizu dd ?
386.         trebaLiSpajatiIliRazdvajati dd ?
387.         sredinaNiza dd ?
388.         stogSPodacimaTrebaLiPetljaRazdvajatiIliSpajatiNizove:
389.             repeat 32768
390.                 dd 0
391.             end repeat
392.
393.
394.         section '.idata' data readable import
395.             library msvcrt, 'msvcrt.dll' ; "msvcrt.dll" je Microsoft Visual C
Runtime Library, dostupna u "C:\Windows\System32\msvcrt.dll" na Windows 98
i novijim.
396.             import
msvcrt, printf, 'printf', system, 'system', exit, 'exit', scanf, 'scanf', clock, 'clock'
397.         AsmEnd

```

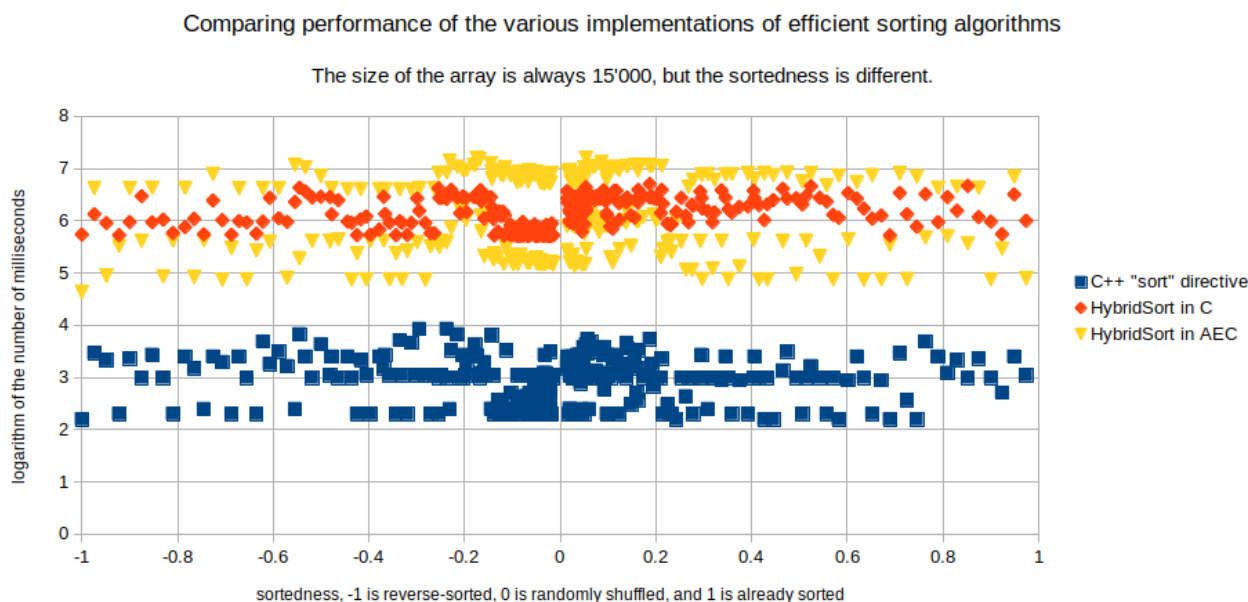
Prije mjerenja, autor je očekivao da će rezultat izgledati približno ovako:



Naime, asemblerski kod koji ispisuje AEC do sada je uvijek bio znatno sporiji od asemblerskog koda što ga ispisuju kvalitetni besplatni C-ovi kompileri, i zato je plava crta iznad crvene. MergeSort bi trebao jednako trajati bez obzira na to kako je niz poredan, i zato većinu crvene i plave krivulje čini horizontalna ravna crta. QuickSort bi u najboljem slučaju, kad je razvrstanost niza približno nula, trebao biti brži od MergeSorta. U stvari, vidjeli smo da je u tom slučaju QuickSort implementiran u C-u neznatno brži od C++-ove naredbe `sort`, zato se narančasta i crvena krivulja dodiruju u sredini. A dodiruju se i na kraju, jer zašto bi MergeSort implementiran u C-u bio sporiji od C++-ove naredbe `sort` kad je niz približno poredan, kad smo vidjeli da se tamo C++-ova naredba `sort` ne ponaša posve optimalno? No, rezultati eksperimenta nisu bili ni približno takvi kakve predviđa ovo teorijsko razmatranje. Program koji je kontrolirao eksperiment dostupan je na autorovom GitHub profilu, pisan je u C++-u i pri pisanju autor je kombinirao tehnike objektivno orijentiranog programiranja i sistemskog programiranja¹⁶. Dostupni su i rezultati koje je taj program za kontrolu tog eksperimenta, nakon što se oko sat vremena vrtio na autorovom

¹⁶ Na ovoj adresi, može se pokrenuti na Linuxu (i vjerojatno se može na Windowsima koristeći Cygwin ili neki sličan compiler, obični C++-ov compiler ne može ga pokrenuti na Windowsima):
<https://github.com/FlatAssembler/ArithmeticExpressionCompiler/blob/master/HybridSort/tester3.cpp>

laptopu, ispisao na ekran, prije no što su kopirani u LibreOffice¹⁷. Ovako izgleda dijagram dobiven obradom eksperimentalnih podataka u LibreOfficeu:



Nije očito kako protumačiti ovakav rezultat. Prvo što upada u oči je da asemblerski kôd koji je ispisao CLANG compiler za C¹⁸ nije brži od asemblerskog koda koji je ispisao moj compiler za AEC, no kao da je precizniji u upravljanju procesorskim vremenom (manja je razlika između gornje crvene točke i donje crvene točke na istoj x-koordinati no što je razlika između gornje narančaste i donje narančaste točke). To bi možda bilo zato što CLANG ispisuje asemblerski kôd predviđen za moderni 64-bitni Intel Core i5 procesor kakav je u mom laptopu, a compiler za AEC pokušava ispisati asemblerski kôd koji će biti kompatibilan i s jako starim procesorima iz obitelji x86, to jest, on ne koristi asemblerske naredbe koje su uvedene nakon i486 i ne ustručava se koristiti naredbe koje nije uputno koristiti ako se pišu programi namijenjeni izvršavanju na novim procesorima. Na taj način on prisiljava procesor u autorovom laptopu da se stalno prebacuje iz načina rada u kojem simulira zastarjele procesore i načina rada potrebnog da bi, recimo, kontaktirao grafičku karticu radi ispisivanja nečeg na ekran (jer su driversi za grafičku karticu, naravno, pisani za novije procesore i očekuju da je on u normalnom načinu rada). Naravno da je teško predvidjeti koliko će procesoru trebati vremena da nešto napravi ako se stalno mora prebacivati iz jednog načina rada u drugi. Također upada u oči to da se čini da se MergeSort ipak ne izvršava uvijek jednako brzo, čak ni u C-u. Autor ovog teksta smatra da je jasno samo po sebi da će se, u gornje izlistanom programu, unutrašnja petlja u MergeSortu (231. do 288. redak) uvijek odvrtni jednak broj puta, no izgleda da se to ne prevodi u to da će se program uvijek izvršiti jednako brzo. Također nije očito, i, u stvari, čini se da nije tako, da je QuickSort u najboljem slučaju brži od MergeSorta. Najbolje objašnjenje za ogromnu razliku u brzini izvršavanja između C++-ove `sort` naredbe i algoritma hibridnog razvrstavanja implementiranog u C-u vjerojatno je to da C++-ov `sort` ne pokušava raditi s decimalnim brojevima da predvidi koji će algoritam razvrstavanja od onih koji su uprogramirani u njega brže završiti, te da rad s decimalnim brojevima usporava taj program napisan u C-u mnogo više no što ga smisleni odabir algoritma koji će se koristiti ubrzava.

Zaključak

Naivna implementacija genetskog algoritma ne daje ni približno točnu formulu za predviđanje koliko će usporedbi QuickSort napraviti na temelju broja elemenata u nizu i razvrstanosti niza. Takva formula omogućavala bi da se vrijeme potrebno QuickSortu da razvrsta neki niz predvidi u

¹⁷ Na ovoj adresi: https://raw.githubusercontent.com/FlatAssembler/ArithmeticExpressionCompiler/master/HybridSort/TestResults/output_of_tester3.txt

¹⁸ Dostupan na autorovom GitHub profilu: <https://github.com/FlatAssembler/ArithmeticExpressionCompiler/blob/master/HybridSort/hsort.s>

linearnom vremenu. To bi vjerojatno bilo korisno za schedulere, i trebali bismo je nastaviti tražiti. Smatram da je najupitnija pretpostavka koju sam napravio da je izvedem to da QuickSortu u najgorem slučaju treba približno n^2 usporedbi. Iz iskustva mi se čini da taj broj nikad ne prelazi $n^2/2$. Možda bi u onoj dugačkoj formuli $(\ln(n) - \ln(\ln(n))) \cdot C_2 \cdot |polinom|$ trebalo zamijeniti s $(\ln(n) - \ln(\ln(n)) - C_3) \cdot C_2 \cdot |polinom|$, gdje bi $C_3 \approx \ln(2) \approx 0.69$. Naravno, za to bi trebalo dosta izmijeniti onaj mekano programirani program.