Se da urmatoarea secventa de cod:

```
int aux, i, j;
int *v, n;      /* Alocate in alta parte */

for (i = 0; i < n; i++)
   for (j = i + 1; j < n; j++)
       if (v[i] > v[j]) {
           aux  = v[i];
           v[i] = v[j];
           v[j] = aux;
       }
```

Tradusa in limbaj de asamblare pentru x86 (optimizat), ea arata in felul urmator:

```
(00)         v       equ [esp - 12]
(01)         n       equ [esp - 8]
(02)
(03)         mov     ebx, v
(04)         mov     ecx, n
(05)         mov     esi, 0
(06)         ; edi = esi + 1
(07)_fori:   mov     edi, esi
(07)         add     edi, 1
(08)         ; boundary check preliminar
(09)         cmp     edi, ecx
(0A)         jge     _skipj
```

```
(0B)_forj:   mov     eax, [ebx + esi]
(0C)         mov     edx, [ebx + edi]
(0D)         cmp     eax, edx
(0E)         jle     _skip
(0F)         mov     [ebx + esi], edx
(10)         mov     [ebx + edi], eax
(11)_skip:   inc     edi
(12)         cmp     edi, ecx
(13)         jl      _forj
(14)_skipj:  inc     esi
(15)         cmp     esi, ecx
(16)         jl      _fori
```

Se cer urmatoarele:

1. Identificati instructiunile care acceseaza in mod explicit memoria.

2. Stiind ca procesorul implementeaza un algoritm de branch prediction dupa cum urmeaza:

- *jump target backward? predict branch taken*
- *jump target forward? predict branch not taken*

Sa se calculeze numarul de branch-uri prezise corect, si numarul de branch-uri prezise gresit pe urmatoarele date de intrare:

```
v = {7, 8, 5}
n = 3
```

3. Calculati timpul/program pentru codul de mai sus, stiind urmatoarele:

- frecventa procesorului = 1 GHz
- numarul de cicli in care se executa instructiunile:

instructiune	**cicli**
`mov` registru-memorie	5
`mov` imediat-memorie	3
`mov` registru-registru	1
`mov` registru-imediat	1

`cmp`	1
`add`	1
branch	20 if mispredicted, 2 if predicted correctly

- datele primite ca input:

```
v = {1, 2, 3, 4, 5}
n = 5
```

4. Presupunem doua ipoteze:

- cea in care imbunatatim timpul urmatoarelor instructiuni:

`mov` reg-mem **3**

`mov` imediat-mem **2**

- cea in care imbunatatim timpul urmatoarei instructiuni:

branch **10** if mispredicted, **1** if predicted correctly

Argumentati in care din cele doua ipoteze se obtine un speedup mai mare, pe exemplul de input dat la subpunctul 3.

Rezolvari:

1. Erau 6 instructiuni care accesau explicit memoria, si anume cele care aveau unul din operanzi cu paranteze patrate. Ele se aflau pe liniile (03), (04), (0B), (0C), (0F) si (10). Pe liniile (00) si (01) nu erau instructiuni, ci pur si simplu niste directive de preprocesor care aveau rolul sa expandeze simbolurile “v” si “n” de pe randurile (03) si (04) in adresele de memorie corespunzatoare.

2.

```
ecx = 3
esi = 0
edi = 1
(0A) not taken (corect)
eax = 7
ebx = 8
(0E) taken (gresit)
edi = 2
(13) taken (corect)
eax = 7
ebx = 5
(0E) not taken (corect)
v = {5, 8, 7}
edi = 3
(13) not taken (gresit)
esi = 1
```

```
(16) taken (corect)
edi = 2
(0A) not taken (corect)
eax = 8
ebx = 7
(0E) not taken (corect)
v = {5, 7, 8}
edi = 3
(13) not taken (gresit)
esi = 2
(16) taken (corect)
edi = 3
(0A) taken (gresit)
esi = 3
(16) not taken (gresit)

Per total: 7 corecte, 5 gresite
```

3. Trebuia urmarita nu foarte atent executia. Era suficient daca se faceau urmatoarele observatii:

- algoritmul de branch prediction prevede ca buclele sunt dese, iar skip-urile sunt rare.
- nu intamplator, destinatiile branch-urilor au fost denumite `_fori`, `_forj`, `_skip`, `_skipj`. ca sa sugereze care din tinte sunt inainte si care inapoi, si cum vor fi ele prezise.
- cu alte cuvinte, buclele vor fi prezise mereu corect, mai putin la ultima iteratie.
- cat despre skip-uri, primul din ele, `_skipj`, pune probleme doar atunci cand i-ul este egal cu n-1, iar al doilea, `_skip`, va fi prezis prost de fiecare data, fiindca vectorul este deja sortat.

Urmatorul pas era urmarirea executiei programului (nu necesita neaparat competente de CN chestia asta, motiv pentru care nu a contat prea mult la punctare rezultatul obtinut):

- exista o sectiune constanta la inceput, inainte de prima bucla, formata din
  2 x movrm***(vezi legenda)** + 1
- pentru i = 0: (2 x movrm + 3 + branchx + branchv) x 3 +
  (2 x movrm + 3 + 2 x branchx) +
  2 + branch
- pentru i = 1: (2 x movrm + 3 + branchx + branchv) x 2 +
  (2 x movrm + 3 + 2 x branchx) +
  2 + branch
- pentru i = 2: (2 x movrm + 3 + branchx + branchv) x 1 +
  (2 x movrm + 3 + 2 x branchx) +
  2 + branch
- pentru i = 3: (2 x movrm + 3 + 2 x branchx) +
  2 + branch

- pentru i = 4: 3 + branch

Bla bla calcule => numarul de instructiuni din secventa de program este format din:

22 x movrm +
15 x branchx +
6 x branchv +
46
=> 89 de instructiuni in total.

Ideea nu era neaparat sa se urmareasca executia si sa se obtina aceste numere, ci niste numere pentru fiecare instructiune. Ce nu era ok era sa considerati ca fiecare instructiune s-ar fi executat o singura data, ca si cand nu ar fi fost loop-uri.
Inlocuind costurile movrm cu 5, branchx cu 20 si branchv cu 2, se obtine un numar de 468 de cicli de ceas.

Timp/program = Timp/ciclu * Instructiuni/program * Cicli/instructiune

Va trebui sa facem o medie ponderata a tuturor instructiunilor, de bine ce tocmai le-am descoperit ponderile.

4. Erau doua metode de rezolvare, fie se inlocuiau costurile noi in aceeasi formula ca mai sus, si apoi se analizau noile costuri obtinute (adica 424 cicli pentru primul caz, si 312 cicli pentru al doilea), fie se calculau ponderile. ATENTIE: acestea trebuiau calculate in cicli de ceas, si nu in numar de instructiuni! Cu alte cuvinte, in cazul de fata, desi procesorul executa mai multe instructiuni de tip movrm (22) decat branch-uri prezise prost (15), acestea din urma petrec mai mult timp din timpul procesorului.
movrm = 23.5% (110 / 468)
branchx = 64.1% (300 / 468)
branchv = 2.5% (12 / 468)
Si se aplica legea lui Amdahl stiind ca instructiunile movrm erau de 5/3 = 1.66 ori mai rapide, iar branch-urile erau de 2 ori mai rapide.

Speedup pentru cazul a)
S1 = 1 / (0.235 / 1.66 + (1 - 0.235))
= 1.103
= 468 / 424
Speedup pentru cazul b)
S2 = 1 / ((0.641 + 0.025) / 2 + (1 - 0.641 - 0.025))
= 1.5
= 468 / 312

Evident, indiferent de metoda folosita ar fi trebuit sa se ajunga la acelasi speedup si aceeasi concluzie.

***Legenda:**
movrm = mov registru-memorie
branchx = branch prezis prost
branchv = branch prezis bine

Examen 11 iunie 2014 (ziua 2 problema 2)

Se considera urmatoarea secventa de cod, care aplica polinomul $\mathbf{x^5 + x^3 + x + 1}$ peste un vector primit ca input:

```
int aux5, aux3, aux;
int v[N];      /* initializat cu date random in intervalul [0 -> M-1] in alta parte */
int precomp[M]; /* initializat cu -1 */

for (i = 0; i < N; i++) {
   aux = v[i];
   if (precomp[aux] == -1) {
       aux5 = aux * aux * aux * aux * aux;
       aux3 = aux5 / aux / aux;
       precomp[aux] = aux5 + aux3 + aux + 1;
   }
   v[i] = precomp[aux];
}
```

Pentru a imbunatati performanta codului de mai sus, vom presupune ca datele de intrare sunt marginite la intervalul **[0 -> M-1]**, si vom folosi o memorie auxiliara, de dimensiune **M**, in care vom stoca valoarea precomputata a polinomului, pe care o putem folosi direct, mai apoi, in loc sa recalculam de fiecare data valoarea lui.
Secventa de cod, tradusa in limbaj de asamblare pentru x86, arata in felul urmator:

```
(00)       v       equ [esp - 12]
(01)       precomp equ [esp - 16]
(02)       N       equ 100
(03)
(04)       mov     esi, 0
(05)       mov     ebx, v
(06)       mov     ecx, precomp
(07)_for:  mov     eax, [ebx + esi]
(08)       mov     edi, [ecx + eax]
(09)       cmp     edi, -1
(0A)       jnz     _skip
(0B)       push    eax ; salvez aux
(0C)       mov     edi, eax
(0D)       mul     eax ; edx:eax = aux ^ 2
(0E)       mul     eax ; edx:eax = aux ^ 4
(0F)       mul     edi ; edx:eax = aux ^ 5
(10)       push    eax ; salvez aux ^ 5
(11)       div     edi ; eax = aux ^ 4
(12)       div     edi ; eax = aux ^ 3
(13)       mov     edi, eax
(14)       pop     eax ; eax = aux ^ 5
(15)       add     edi, eax
(16)       ; edi = aux ^ 5 + aux ^ 3
(17)       inc     edi
(18)       ; edi = aux ^ 5 + aux ^ 3 + 1
(19)       pop     eax ; eax = aux
(1A)       mov     [ecx + eax], edi
(1B)_skip: mov     eax, [ecx + eax]
(1C)       mov     [ebx + esi], eax
(1D)       inc     esi
(1E)       cmp     esi, N
(1F)       jl      _for
```

Se cer urmatoarele:

1 (3p). Identificati, in varianta a doua a codului, instructiunile care acceseaza in mod explicit memoria.

2 (3p). Procesorul are un algoritm de branch prediction dinamic (global saturating counter) ce foloseste un FSM cu patru stari:

- *strongly taken* - *weakly taken* - *weakly not taken* - *strongly not taken*

FSM-ul porneste din starea "*strongly not taken*" si, pentru o predictie de tip "(not) taken" corecta, merge in directia starii "*strongly (not) taken*", iar pentru o predictie gresita, merge in directia opusa.
Stabiliti numarul de branch-uri prezise corect si numarul de branch-uri prezise gresit, precum si starile prin care trece branch predictorul, in urma executiei celei de-a doua variante de cod, cu urmatorul set de date de intrare:

```
M = 10              N = 3              v[] = {1, 1, 2}
```

3 (2p). Calculati timpul/program pentru o rulare a codului de mai sus (varianta a doua) cu **N = 1000** si **M = 10**, presupunand ca toate branch-urile vor fi prezise corect, si ca datele vor avea o distributie uniforma (in vector vor aparea toate valorile de la 0 la 9). De asemenea, se dau urmatoarele informatii despre procesor:

Frecventa ceasului - 1 GHz

Instructiune	Numar de cicli	Instructiune	Numar de cicli	Instructiune	Numar de cicli
`mov` registru-imediat	1	`mul`	100	cmp	1
`mov` registru-registru	1	`div`	300	`push`/`pop`	10
`mov` registru-memorie	10	`add`/`inc`	1	`branch`	30

4 (2p). Stiind ca avem **M** fixat la valoarea **10**, sa se determine, folosind legea lui Amdahl, dimensiunea minima **N** a vectorului, astfel incat codul cu imbunatatirea adusa, cea prin aducerea datelor precomputate din memorie, sa obtina un speedup de minim **10** ori mai mare decat acelasi cod, in care toate datele sunt calculate, si nu aduse din memorie. Se poate presupune, pentru simplitate, ca toate branch-urile sunt prezise in mod corect, si se poate ignora overhead-ul instructiunilor din afara buclei.

Rezolvari:

1. La fel ca la cealalta problema, instructiunile care accesau memoria erau cele cu paranteze patrate, impreuna cu push-urile si pop-urile. Asadar, (05), (06), (07), (08), (0B), (11), (15), (19), (1A), (1B), (1C). Deci, in total, 11 instructiuni.

2.
FSM strongly not taken => predict not taken
precomp[1] = -1
(0A) not taken (correctly predicted)
FSM strongly not taken => predict not taken
(1F) taken (incorrectly predicted)
FSM weakly not taken => predict not taken
precomp[1] = 3
(0A) taken (predicted incorrectly)
FSM weakly taken => predict taken
(1F) taken (correctly predicted)
FSM strongly taken => predict taken
precomp[2] = -1
(0A) not taken (incorrectly predicted)
FSM weakly taken => predict taken
(1F) not taken (incorrectly predicted)
In total, 2 corecte, 4 gresite.

3. In primul rand, trebuie sa pornim de la observatia ca, pentru un set de date asa mare, computarea efectiva se va face numai de 10 ori, dupa care se va face skip, luandu-se mereu datele precomputate din memorie. Asadar, putem calcula timpul pentru o computatie, il inmultim cu 10, apoi calculam timpul pentru aducerea din memorie a unei precomputari, si il inmultim cu N - 10. In final, trebuie sa adaugam si overhead-ul de 3 instructiuni de la inceput.

Initial (in afara buclei): 2 x movrm + 1 x movri = 21 de cicli

Computare (de 10 ori):
5 x movrm - 50 cicli
2 x cmp - 2 cicli
2 x branch - 60 cicli
4 x push/pop - 40 cicli
2 x movrr - 2 cicli
3 x mul - 600 cicli
2 x div - 600 cicli
3 x add - 3 cicli

Aducere din memorie (N - 10 ori):
4 x movrm - 40 cicli
2 x cmp - 2 cicli
2 x branch - 60 cicli
1 x add - 1 ciclu

In total, avem 1357 de cicli pentru computare, si 103 cicli pentru aducere din memorie.

Inmultind cu ponderile (10 pentru computare, 990 pentru aducere din memorie), obtinem un numar total de cicli per program egal cu 21 + 1357 * 10 + 103 * 990 = 115 561 cicli.

4. Trebuie sa observam ca imbunatatirea, conform legii lui Amdahl, consta in aducerea din memorie. Factorul de imbunatatire este de 1357/103, adica aproximativ 13 ori. De asemenea, pentru un set de date de intrare de dimensiune N, imbunatatirea se aplica asupra a N - 10 numere, restul de 10 fiind imposibil de imbunatatit.
Acestea fiind zise, formula se aplica astfel:
S = 1 / (((N - 10)/N) * (1/13) + 10/N)
Se dorea un speedup de minim 10:
(N - 10)/13N + 10/N = 1/10
(N - 10)/13N + 130/13N = 1/10
(N + 120)/13N = 1/10
10N + 1200 = 13N
3N = 1200
N = 400