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     ebx, [ebx + edi]
(0D)         cmp     eax, ebx
(0E)         jle     _skip
(0F)         mov     [ebx + esi], ebx
(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:
movrm = 24.7%
branchx = 16.8%
branchv = 6.7%
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. Evident, 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