Arbori indexați binar (AIB).

Materialul de față se va concentra mai puțin pe structura internă a `AIB`-urilor și mai mult pe modul lor de folosire.

Arborii indexați binar reprezintă o structură de date utilă pentru a rula rapid anumite operații pe șiruri, de bază fiind: actualizare a unui element și interogare pentru un prefix. În anumite cazuri aceste operații pot fi adaptate și pentru secvențe oarecare, nu neapărat elemente individuale sau prefixe.

Afirmăm că `AIB` sunt o structură de date deoarece dintre toate secvențele unui șir ei precalculează informații pentru o parte dintre acestea și pe baza acestor informații se pot reconstitui apoi informații din orice altă secvență.

Memoria necesară este de ordin n (pentru fiecare element se stochează date despre o singură secvență care se termină în el), deci pentru acest lucru este necesar un șir de lungimea celui dat.

Operațiile de interogare și de actualizare se fac în timp logaritmic. Adică la o interogare pe prefix vom folosi maxim log secvențe dintre cele la care avem calculată informația iar la o actualizare vom modifica de asemenea maxim `log` dintre aceste secvențe.

O structură alternativă pentru lucrul cu informații despre secvențe o reprezintă arborii de intervale (`AINT`).
`AINT` necesită tot memorie de ordin `n` dar în realitate constanta este mai mare (dacă la `AIB` ne trebuie exact `n` memorie suplimentară, la `AINT` memorăm informații despre aproximativ `2*n` intervale dar la implementare este necesară o structură mai mare, pentru siguranță până la `4*n` componente).
Arborii de intervale acceptă în general operațiile pe care le acceptă și arborii indexați binar dar și altele neacceptate de `AIB`. În schimb, atunci când o operație este acceptată și de `AIB` se preferă folosirea acestora din urmă (nu numai pentru necesarul de memorie dar și pentru viteză, la `AIB` unele lucruri se fac ceva mai repede pentru că implementarea se poate folosi mai mult de operații pe biți iar la `AINT` unele lucruri merg ușor mai lent și datorită apelurilor recursive necesare în număr mare).

Să prezentăm acum mai detaliat lucruri utile despre `AIB`.

Enunț: Se dă un șir `V` de `n` valori, inițial toate nule. Asupra lui se pot efectua operații de două tipuri: mărirea elementului de pe poziția `p` cu valoarea val (adică `V[p] += val`, codificată de noi `1 p val`), respectiv interogarea pentru sumă a secvenței de la poziția `st` la poziția `dr` (notată `2 st dr`).

În această problemă interogările și actualizările se pot face intercalat, la fiecare interogare trebuie să răspundem ținând cont de configurația șirului dată de actualizările precedente.

Observații:

- Simularea pe un vector obișnuit permite actualizarea în timp constant și interogarea în timp liniar (ne trebuie un for de la `st` la `dr` la fiecare interogare);

- Nici tehnica sumelor parțiale nu ne oferă eficiență ca și timp pe ambele operații deoarece acestea trebuie actualizate după fiecare operație de tip `1` (toate sumele de la poziția `p` pana la poziția `n`, deci și aici ar trebui timp de calcul de ordin `n`).

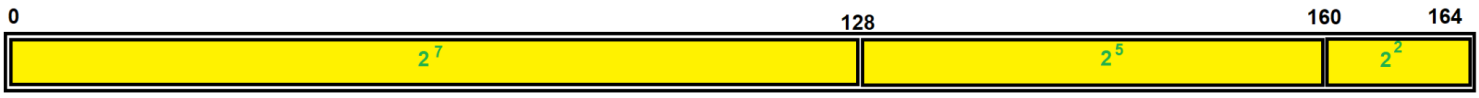
Să vedem acum cum putem obține timp de calcul logaritmic pentru ambele operații prin utilizarea arborilor indexați binar.

Trebuie să avem în minte soluția cu sume parțiale, numai că noi nu vom păstra toate sumele parțiale ci le vom obține când avem nevoie pe baza sumelor din secvențele stocate în structură.

Ne vom folosi de scrierea unui număr în baza `2`. Pentru exemplificare, să considerăm numărul `164` pentru care scrierea în baza `2` este `10100100`.
`164` ca sumă de puteri distincte de `2` se scrie: $2^7 + 2^5 + 2^2$.
Noi pe `164` ni-l imaginăm indice în vectorul `V` și totodată în structura `A`.
Spuneam că trebuie să avem în minte problema sumelor parțiale deoarece noi dacă dorim suma din `V` dintre două poziții de fapt o vom obține ca diferență dintre două sume parțiale. Deci ne concentrăm spre ce avem de făcut pentru a obține suma din `V` dintre indicii `1` și `164`.
În structura de date `A`*, la poziția* `164` *vom păstra suma din secvența din* `V` *cu următoarele proprietăți: se termină la poziția* `164`*; are lungimea* 2^2 *(adică ultima putere de* `2` *care apare în scrierea lui* `164` *în vaza* `2`*).*
Observăm că dacă scădem din `164` această putere de `2` (echivalent cu a șterge ultimul bit de `1` din scrierea în baza `2` a lui `164`), valoarea obținută este $164 - 2^2 = 160$, `10100000`.
Acum continuăm cu `160`. Deci: considerăm suma din `V` pentru secvența de lungime 2^5 care se termină la indicele `160`. Această valoare va fi chiar `A[160]`. Ștergem iarăși ultimul bit de `1` și ajungem la `160-32 = 128`. Acest număr are un singur bit de `1`, deci `A[128]` va reprezenta suma din `V` de lungime $128 = 2^7$ terminată la poziția `128`, adică practic aceea care începe la poziția `1` (observăm că în `AIB` la poziții putere de `2` avem chiar suma din secvența din vectorul original care începe la poziția `1` și se termină la acea poziție).
Expresia `i -= (i&-i)` are ca efect ștergerea ultimului bit de `1` din scrierea binară a lui `i` (nu vom detalia aici acest lucru dar îl puteți fixa fie studiind materialul de la capitolul dedicat operațiilor pe biți fie, mai bine, simulând efectiv operațiile pe un exemplu concret).
Operația de interogare, o putem realiza cu secvența de mai jos.

`query(p)` : oferă în timp logaritmic suma din `V` pentru elementele dintre pozițiile `1` și `p`.

```
suma = 0;
for (i = p; i > 0; i-=(i&-i))
    suma += A[i];
```



$164 = 2^7 + 2^5 + 2^2$

Suma din V de la pozitia 1 la pozitia 164 = A[164] + A[160] + A[128]

Așadar pentru a obține suma numerelor din V dintre pozițiile 1 și 164 este necesar să adunăm doar trei elemente din A. Întrucât la fiecare pas al repetiției practic se elimină un 1 din scrierea binară a indicelui curent, rezultă că numărul de pași este maxim egal cu numărul maxim de poziții necesare la reprezentarea binară a poziției dată, adică $\log_2$ din aceasta.
Evident că pentru a beneficia de acest lucru ar trebui să avem grijă să modificăm corespunzător pe A atunci când actualizăm elemente din V.
La o operație de adăugare a valorii x la V[p], noi ar trebui să mărim cu x toate valorile din A care corespund la secvențe ce conțin poziția p. Acest lucru se obține cu un for similar cu acela de sus dar care mărește valoarea indicelui.
Update (p, x) : actualizează în timp logaritmic structura pentru a nota că elementul de pe poziția p **a fost mărit** cu valoarea x (adică adună x la cel mult log elemente din A) **.**

```
for (i = p; i <= n; i+=(i&-i)) /// i va ajunge indicele de final al fiecarui interval care contine pozitia p (dintre cele necesare AIB-ului)
  A[i] += val;
```
Semnificatie: daca am marit cu val un element al lui V creste cu val suma din orice interval al AIB - ului care continne elementul marit cu val

Diferența la implementare este că mergem acum cu i crescător, deci până la n și că avem + în loc de − la modificarea lui i.
Am specificat acest detaliu de implementare pentru că de cele mai multe ori se formează automatism la scrierea acestui cod (el fiind foarte scurt).

Să luăm un exemplu pentru a înțelege însă mai bine ce se întâmplă.
Avem un șir V de lungime n și facem update cu valoarea x la poziția p.
Să presupunem n = 84 și p = 44.

Aici la fiecare pas se transformă în 0 secvența de valori 1 vecine de la cel mai puțin semnificativ 1 spre stânga și prima poziție 0 întâlnită se face 1. Dacă am privi acum invers dinspre această valoare. Numărul de pași este logaritmic pentru că practic și aici se curăță valori 1 (dispare cel puțin un 1 iar unul nou se adaugă mai la stânga, deci nu putem avea număr de pași mai mare ca numărul de poziții ale scrierii binare a lui n).

Privind invers, dinspre valoarea construită spre cea din care am construit, observăm că poziția p face parte din secvența reprezentată de poziția din AIB la care am ajuns.

Concluzionăm:

- Facem query de sumă pentru secvența de la 1 la p din V adunând maxim log valori din A cu prima secvență de cod prezentată.
- Facem update la un element din V actualizând maxim log elemente din A cu a doua secvență de cod prezentată.

- La implementare vectorul `V` nu este necesar, dar trebuie să îl avem în minte la orice operație pe care o facem pe `A`.
- Dacă dorim suma din secvența cu indici cuprinși între `st` și `dr`, facem două interogări: `query(dr) - query(st-1)`.
- **Structura funcționează și pentru maxime pe prefix, dar atenție, doar dacă la update valoarea elementelor crește față de cea anterioară (similar pentru minim când știm că valorile scad mereu).**

În acest caz avem în `A[i]` maximul din secvența cu finalul la poziția `i` și de lungime cea mai mică putere de `2` care apare în scrierea binară a lui `i`. Practic la implementare în loc de `suma += A[i]` vom avea `sol = max(sol, A[i])` în query respectiv `A[i] = max(A[i], x)` în update (aici am presupus că operația de update nu înseamnă că elementul din `V` crește cu `x` ca la sumă ci că el devine `x`, presupunând că `x` este o valoare mai mare decât cea care era înainte în `V[p]`).

Chiar dacă este puțin redundant să reluăm, dar ca o concluzie, noi trebuie să ne gândim că la query obțin de fapt informația pe prefix iar la update actualizez toate intervalele necesare la un query pentru un prefix mai mare sau egal cu poziția pe care fac update.

Am prezentat operațiile pe `AIB` pornind de la un șir cu toate valorile nule. Putem porni și de la un șir cu altă configurație dar trebuie ca la început să îl transformăm mai întâi în `AIB` (cu `n` operații de update, câte una pentru fiecare element, deci timp de ordin $nlog_2n$).

Probleme rezolvate.

1. Se dă o permutare a mulțimii `1`, `2` ...`n`,memorată într-un vector `V` cu `n` elemente. Determinați câte inversiuni are în total. Numărul de inversiuni pentru o poziție `i` esre egal cu numărul de valori valori care se află pe poziții mai mici decât `i` și care totodată sunt mai mari decât `V[i]`.

Exemplu: `2 5 1 6 4 3` are `7` inversiuni (`1` are în față două elemente mai mari decât el și `4` are două iar `3` are trei).

Soluția `1`.
Cu două foruri, încercăm elementele fiecare cu fiecare (pe structura algoritmului de sortare prin comparare).

```
for (i=2;i<=n;i++)
    for (j=1;j<i;j++)
        if (V[j] > V[i])
            inversiuni++;
```

Timpul de calcul este de ordin n^2 deci algoritmul nu este practic pentru valori mai mari ale lui `n`.

Soluția `2`.
Este tot una cu timp de calcul de ordin n^2 însă pornind de la ea vom putea optimiza ulterior. Să o prezentăm mai întâi.
Facem următoarea observație: numărul total de perechi de indici este `n*(n-1)/2`. Putem să numărăm, ca în enunț la fiecare poziție câte are în față cu valori mai mari sau putem calcula câte elemente are în față cu valori

mai mici și scădem din numărul total de poziții. Pentru a putea explica mai ușor, vom merge pe varianta a doua mai departe.

Vom folosi un vector de frecvență `F` și când ajungem la poziția `i` în `V` facem `F[ v[i] ] = 1;`

```
for (i=1;i<=n;i++)
    F[ V[i] ] = 1;
```

Ce avem însă în vectorul `F` înainte să marcăm pe `V[i]` ? Nu uităm că noi dorim să aflăm pentru o poziție câte elemente aflate mai în față sunt mai mici. Să analizăm exemplul anterior, când ajungem cu `i = 5`. (Adică la elementul cu valoarea `4`).
În vectorul `F` sunt marcate cu `1` doar elementele întâlnite înainte de poziția `i`, adică `1`, `2`, `5` și `6`. Noi pentru `4` ar trebui să obținem că avem două mai mici, adică pe `1` și pe `2` (`3` nu ne interesează că el apare în vector după poziția `4`). Analizând figura de mai jos observăm că pe noi ne interesează numarul de valori `1` de pe poziții de la `1` la `4` în momentul întâlnirii lui `4` .

Indice	1	2	3	4	5	6
F	1	1			1	1

Astfel, o primă variantă ar fi să mai adăugăm un for înainte să îl marcăm pe `4` în `F`, analizând însă doar indici mai mici decât `4`.

```
sol = n*(n-1)/2;
for (i=1;i<=n;i++) {
    for (j=1;j<V[i];j++)
        if (F[j] == 1)
            sol--;
    F[ V[i] ] = 1;
}
```

Da, acest algoritm este unul cu timpul de calcul n^2 (de fapt, fără o normalizare în prealabil, dacă nu era vorba despre o permutare, aveam timp de calcul de ordin `n * (valoarea maximă)`).

Soluția `3`.
O prezentăm ca o continuare a celei anterioare. Observăm că actualizarea lui `F` înseamnă să mărim un element al său cu `1` iar interogarea este suma pe un prefix din `F`. Așadar suntem exact în conctextul care ne permite restructurarea informațiilor din `F` într-un `AIB` (să îl notăm `A`).
Vom avea:

```
int query(int p) {
    int suma = 0;
    for (int i=p;i;i-=(i&-i))
```

```
        suma += A[i];
    return suma;
}

void update(int p) {
    for (int i=p;i<=n;i+=(i&-i))
        A[i]++;
}

int main () {
    cin>>n;
    for (i=1;i<=n;i++)
        fin>>v[i];

    sol  = n*(n-1)/2;
    for (i=1;i<=n;i++){
        sol -= query(V[i]-1);
        update(V[i]);
    }
    fout<<sol;
}
```

Observăm ușor că vorbim despre timp de calcul de ordin $n*\log_2 n$ din faptul că în repetiție avem o operație de actualizare și una de interogare.

2. **AIB 2d.**

Avem o matrice cu `n` linii și cu `n` coloane care inițial are toate elementele egale cu `0`. Asupra sa se aplică două tipuri de operații:

- Actualizare: `i j x` elementul de la indicii (`i`,`j`) crește cu `x`.
- Interogare: se dau `4` indici `i1`, `j1`, `i2`, `j2` și se cere să aflăm suma din submatricea cu colțul stânga sus (`i1`,`j1`) și dreapta jos (`i2`,`j2`).

Prezentăm mai întâi implementarea (extrem de scurtă) apoi o vom explica.

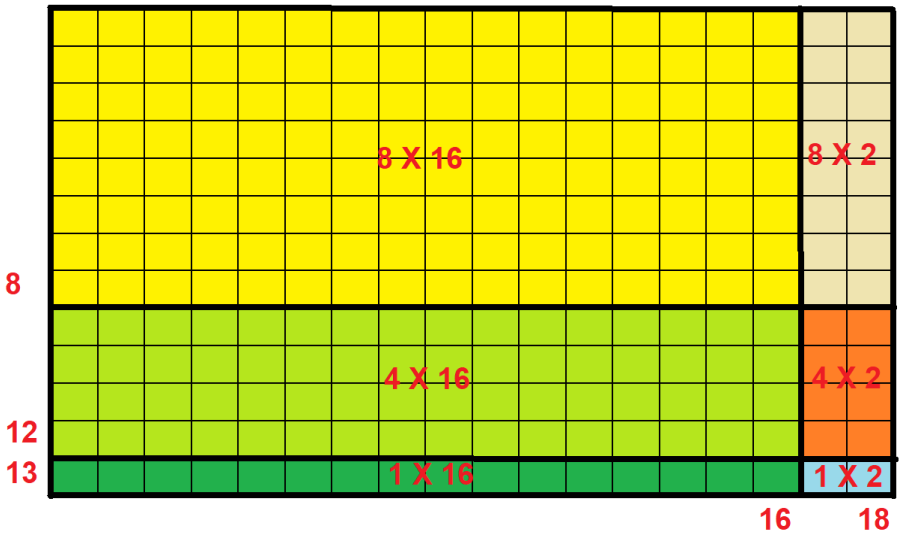
```
int query(int pi, int pj) {
    int suma = 0;
    for (int i=pi;i;i-=(i&-i))
        for (int j=pj;j;j-=(j&-j)
            suma += A[i][j];
    return suma;
}

void update(int pi, int pj, int x) {
    for (int i=pi;i<=n;i+=(i&-i))
        for (int j=pj;j<=m;j+=(j&-j)
            A[i][j] += x;
}
```

În main, la o interogare cu colțul stânga sus `i1,j1` și dreapta jos `i2,j2` răspunsul este:
`query(i2,j2) - query(i1,j2-1) - query(i1-1, j2)+query(i1-1,j1-1);`
Adică exact ca la sume parțiale pe matrice. Adică `query(i,j)` dă suma dintr-o submatrice cu colțul stânga-sus (`1,1`) și dreapta jos (`i,j`) iar pentru o suma pe o submatrice oarecare rezultatul se compune din patru astfel de "sume parțiale".

La `AIB2d` în `A[i][j]` păstrăm suma din submatricea care are colțul dreapta jos în (`i,j`), numărul de linii este cea mai mică putere de `2` care apare în scrierea binară a lui `i` iar numărul de coloane este cea mai mică putere de `2` care apare în descompunerea în factori primi a lui `j`.

De exemplu, la un `query(13, 18)`, cele două foruri adună sumele precalculate din dreptunghiuri ca în figură:



$13 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1$
$18 = 2^4 + 2^1 = 16 + 2$

Practic se face pentru ambele dimensiuni ceea ce se avem la `AIB`-urile pe vector. Atenție că mai sus nu avem reprezentarea matricei `A` ci sunt puse în evidență dreptunghiurile a căror sumă se adună la un `query(13,18)`.
Pe exemplul de mai sus sumele din aceste dreptunghiuri se găsesc în elementele: `A[8][16]`, `A[8][18]`, `A[12][16]`, `A[12][18]`, `A[13][16]`, `A[13][18]`.
Fiecare element din `A` reprezintă așadar o zonă dreptunghiulară cu ambele dimensiuni putere de `2`.
Timpul de calcul este de ordin $\log^2$ atât pentru query cât și pentru update.