Metoda podjeli pa vladaj

Ciljevi:

- upoznati se sa metodom podjeli pa vladaj
- shvatiti logiku rada ove metode
- naučiti implementirati metodu po potrebi
- uvidjeti razloge uvođenja ove metode

Pregled lekcije

„Divide et impera – Podjeli pa (o)vladaj“

Julije Cezar (102 – 44 p. n. e.). Rimski car i vojskovođa.

Ova se metoda temelji na tri koraka. Prvi korak se temelji na tome da se originalni problem podjeli na dva ili više ista problema manjih dimenzija. Nakon toga se kreće u rješavanje novostvorenih problema manjih dimenzija a nakon toga se rješenja potproblema spajaju u rješenja prvotnog problema.

Primjer 1:
ULAZ: Decimalni broj E i prirodni broj P
IZLAZ: EP

Algoritam se temelji na činjenici da vrijedi rekurzivna formula

$$E^P = \begin{cases} E^{P/2} \cdot E^{P/2} \text{ za } P \text{ paran} \\ E^{P/2} \cdot E^{P/2} \cdot E \text{ za } P \text{ neparan} \end{cases}$$

uz rubni uvjet

$E^1=E$,
$E^0=1$

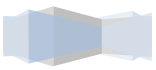
- Funkcija ipow(E,P)
  1. Ako je P<2
  2. Ako je P=0 onda vrati 1
  3. inače vrati E
  4. inače
  5. Ako je P paran
  6. vrati ipow(E, P/2)*ipow(E, P/2)
  7. inače
  8. vrati ipow(E,P/2)*ipow(E,P/2)*E

Primjer 1.1:

```
#include <iostream>
using namespace std;
int ipow(double E, int P) {
    if (P<2)
       if (P==0)
          return 1;
       else
           return E;
    else
    {
        double p = ipow(E,P/2);
        if (P%2)
           return p*p*E;
        else
            return p*p;
    }
}

int main () {

    double E;
    int P;

    cout << "E = ";
    cin >> E;

    do {

       cout << "P = ";
       cin >> P;

    } while (P<0);

    cout << "E^P = " << ipow(E,P) << endl;

    system("pause");

    return 0;
}
```

Pretraživanja i sortiranja temeljena na metodi podijeli pa vladaj

Metoda podijeli pa vladaj između ostalog daje i efikasnije algoritme za pretraživanje i sortiranje, stoga ćemo na sljedećim primjerima pokazati na koji način je ova metoda implementirana kroz neke od algoritama.



Pretpostavimo da pretražujemo polje koje je već sortirano uzlazno. Tada postoji bolji način pretraživanja od onog kojeg smo do sada koristili i zove se **binarno pretraživanje**.

Algoritam binarnog pretraživanja funkcionira na način da se prvo usporedi element u sredini polja s traženom vrijednosti i sada s obzirom na to da li je on manji ili veći od tražene vrijednosti poduzimamo sljedeće korake:

- Ako je manji, onda traženi element (ako se nalazi u polju), mora biti u drugoj polovici polja
- Ako je jednak, onda smo ga pronašli i završavamo
- Ako je veći, onda se traženi element (ako postoji u polju), mora biti u prvoj polovici polja

Program koji pretražuje niz ćemo napraviti rekurzivno. BinSearch će zvati sam sebe s promijenjenim parametrima (i,j), kako je opisano u algoritmu. Kod rubnih uvjeta u ovom slučaju treba biti oprezan pošto se mogu dogoditi dvije situacije koje se izravno rješavaju:

- Ako je i=j, onda se treba provjeriti je li P[i]=k i prema tome vratiti **`true`** ili **`false`**.
- Ako pak je i>j, onda se treba vratiti **`false`**.

Primjer 2:

ULAZ: Broj N, niz A te traženi broj K
IZLAZ: Potvrda (**`true`**/**`false`**) o toma dali je broj pronađen

1. Učitaj N
2. Učitaj niz A
3. Učitaj K
4. Ako je BinSearch(A,K,0,N-1)
5. ispiši "Broj se nalazi u nizu"
6. inače
7. ispiši "Broj se ne nalazi u nizu"

- Funkcija BinSearch(P,K,i,j)
  1. Ako je j-i<=0
  2. Ako je i==j i P[i]==K
  3. vrati true
  4. inače
  5. vrati false
  6. inače
  7. k = (i+j)/2
  8. Ako je (P[k]>K) vrati BinSearch(P,K,i,k)
  9. Ako je (P[k]=K) vrati true
  10. Ako je (P[k]<K) vrati BinSearch(P,K,k,j)

Primjer 2.1:

```
#include <iostream>
using namespace std;

bool BinSearch(float P[], float K, int i, int j) {
     if (j-i <= 0)
```

```
        if (i==j && P[i] == K)
           return true;
        else
            return false;
     else {

         int k = (i+j)/2;

         if (P[k]>K)
            return BinSearch(P,K,i,k-1);
         if (P[k]==K)
            return true;
         if (P[k]<K)
            return BinSearch(P,K,k+1,j);
     }
}

int main () {

    int N, k;
    float A[1000], K;

    do {
       cout << "N = ";
       cin >> N;
    } while (N < 2 && N > 1000);

    for (int i = 0; i < N; i++) {
        cout << "A[" << i
             << "] = ";
        cin >> A[i];
    }

    cout << "K = ";
    cin >> K;

    if (BinSearch(A,K,0,N-1))
       cout << "Broj se nalazi u nizu" << endl;
    else
        cout << "Broj se ne nalazi u nizu" << endl;

    system("pause");

    return 0;
}
```

Kao što se može i zaključiti kod binarnog pretraživanja se tražena vrijednost ne uspoređuje sa svakim elementom polja. U prvom se koraku pretraživanja eliminira pola elemenata, u drugom još pola od te polovice itd.

Neka je $N = 2^k$. Tada nakon prvog koraka ostaje 2^{k-1} elemenata, nakon drugog 2^{k-2} itd. Jasno je da postupak može imati najviše k koraka, a u svakom od njih se vrši jedna usporedba. Kako je $N = 2^k$, vrijedi da je $k = \log_2 N$, pa je najveći broj usporedbi $\log_2 N$.

Napomena 1: Ako polje nije sortirano, algoritam neće raditi!

Napomena 2: Nikada se ne isplati sortirati polje samo da bi se pretraživalo. Sortiranje ima više usporedbi nego slijedno pretraživanje!

Još jedan jako dobar algoritam koji se temelji na metodi podjeli pa vladaj je i sortiranje spajanjem (**Merge sort**). Sortiranje spajanjem je algoritam sortiranja koji se temelji na doslovnoj primjeni metode podijeli pa vladaj.

Osnovni pseudokod sortiranje spajanjem je:

- Ako je polje veće od jednog elementa
  I. Podijeli polje na dvije polovice
  II. Sortiraj svaku polovicu
  III. Spoji sortirane polovice u sortirano polje

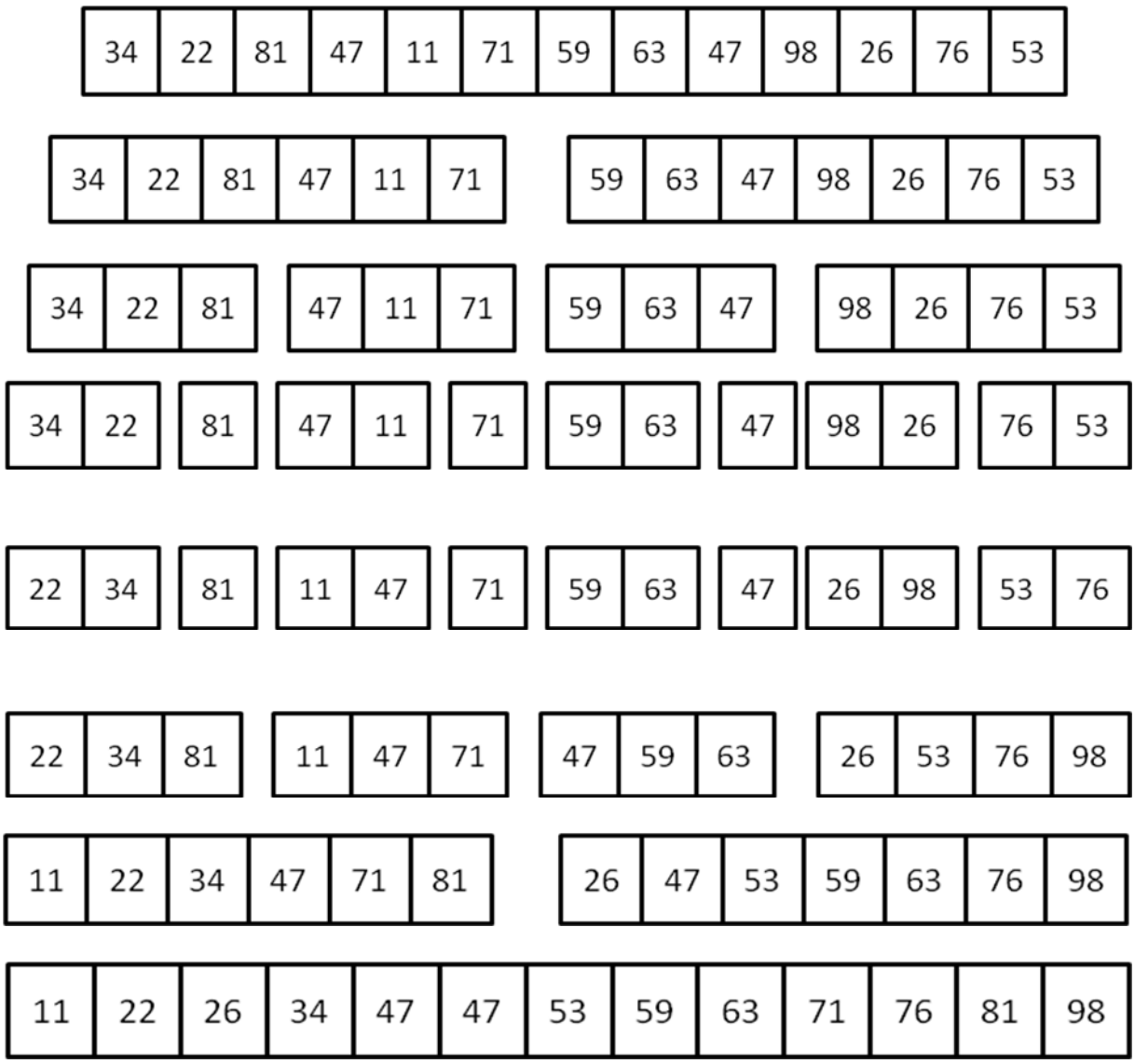
Ali postavlja se pitanje kako se dva sortirana polja `A[i]` i `B[j]` spajaju u jedno sortirano polje `C[i+j]`?

1. Sve dok u oba polja ima još neobrađenih elemenata
   I. Uzmi manji od prvih neobrađenih elemenata u ulaznim poljima i stavi ga na kraj izlaznog polja
   II. Promatraj sljedeći neobrađeni element u polju iz kojeg je uzeti element
2. Prepiši sve preostale neobrađene elemente uz polja u kojem ih još ima

Konkretnije bi izgledalo ovako:

```
1.  I = 0, J = 0, K = 0
2.  Sve dok je I <= i i J <= j radi
3.          Ako je A[I] < B[J]
4.                  C[K] = A[I]
5.                  I = I + 1
6.          inače
7.                  C[K] = B[J]
8.                  J = J + 1
9.          K = K + 1
10. Ako je (I>i)
11.         Sve dok je J<=j
12.                 C[K] = B[J]
13.                 K = K + 1
14.                 J = J +1
15. inače
16.         Sve dok je I<=i
17.                 C[K] = A[I]
18.                 K = K + 1
19.                 I = I +1
```

Sljedeća slika daje primjer sortiranja spajanjem.



Slika 1: Sortiranje spajanjem

I ovaj ćemo algoritam izvesti rekurzivno, s time da se rubni uvjet rekurzije temelji na činjenici da je sortiranje polja od jednog elementa trivijalno i da ga stoga možemo izvesti izravno.

Primjer 3:

- Funkcija MSort(P,I,J)
  1. Ako je I<J
  2. K=(I+J)/2
  3. MSort(P,I,K)
  4. MSort(P,K+1,J)
  5. Spoji(P,I,K,J)

Nadalje, operaciju spajanja treba u jednoj mjeri preraditi tako da radi za dva dijela istog polja. Bit će nam potrebno dodatno polje `B` u koje ćemo privremeno staviti sortirani niz.

Naime, niz ne možemo direktno spajati u polje koje sortiramo jer bismo time "pregazili" podatke koji su nam još potrebni za spajanje.

- Funkcija Spoji(P,i,k,j)
  1. I = i
  2. J = k+1
  3. K = 0
  4. Sve dok je I <= k i J <= j radi
  5. Ako je A[I] < A[J]
  6. B[K] = A[I]
  7. I = I + 1
  8. inače
  9. B[K] = A[J]
  10. J = J + 1
  11. K = K + 1
  12. Ako je (I>k)
  13. Sve dok je J<=j
  14. B[K] = A[J]
  15. K = K + 1
  16. J = J +1
  17. inače
  18. Sve dok je I<=i
  19. B[K] = A[I]
  20. K = K + 1
  21. I = I +1
  22. Za I = 0..j-i radi
  23. A[i+I] = B[i]

Primjer 3.1:

```
#include <iostream>
using namespace std;

void Spoji(float A[],int i,int k,int j) {

      int I=i, J=k+1, K=0;
      float *B = new float [j-i+1];

      while (I<=k && J<=j)
            if (A[I]<=A[J])
               B[K++]=A[I++];
            else
                 B[K++]=A[J++];
      if (I>k)
         while (J<=j)
               B[K++] = A[J++];
      else
          while (I<=k)
                B[K++] = A[I++];
```

```
      for (int I=0;I<=j-i;I++)
          A[i+I]=B[I];

      delete []B;
}

void MSort(float A[],int i, int j) {

      if (i<j) {
         int k=(i+j)/2;
         MSort(A,i,k);
         MSort(A,k+1,j);
         Spoji(A,i,k,j);
      }
}

void MSort(float A[],int N) {
      MSort(A,0,N-1);
}

int main () {

     int N, I;

     do {
        cout << "N = ";
        cin >> N;
     } while (N < 2 && N > 1000);

     float *A = new float [N];

     for (int i = 0; i < N; i++) {
         cout << "A[" << i << "] = ";
         cin >> A[i];
     }

     MSort(A,N);

     for (int i = 0; i < N; i++)
         cout << A[i] << " ";

     cout << endl;

     system("pause");

     delete []A;

     return 0;
}
```

Broj usporedbi i zamjena u ovom slučaju nećemo izračunavati jer bi nam za to trebala "viša matematika".

Broj usporedbi: $N \cdot \log_2 N$ Broj zamjena: $2 \cdot N \cdot \log_2 N$

Problem ovog načina sortiranja je u tome što je potrebno dodatno polje, međutim postoji način sortiranja koji rješava problem potrebe dodatnog polja i zove se **QuickSort**. Naime QuickSort drugačije dijeli polje, zbog čega spajanje, pa onda ni dodatno polje nije potrebno.

Kod QuickSort algoritma polje se ne dijeli onakvo kakvo je, već se prije podjele polja na dva dijela elementi preraspoređuju. Određuje se neka vrijednost, koja se naziva kružni element ili sidro (pivot) a preostali se elementi raspoređuju s obzirom na njega. Elementi koji su manji od njega premještaju se u polju ispred njega, a oni koji su veći od njega se premještaju se iza njega. Nakon što se izvede ova transformacija potrebno je samo još sortirati dijelove polja koji se nalaze ispred i iza kružnog elementa, dok njihovo spajanje nije potrebno.

Isto tako QuickSort algoritam ne ovisi o načinu na koji se bira kružni element. Drugim riječima, sam algoritam sortiranja neće se mijenjati promjenom strategije izbora kružnog elementa. No, strategija izbora kružnog elementa može znatno utjecati na brzinu sortiranja.

Primjer 4:

- Funkcija QuickSort(A,i,j)
  1. Izaberi pivota P
  2. I=0, J=N-1
  3. Sve dok je I<J radi
  4. Sve dok je A[I]<P radi I=I+1
  5. Sve dok je J>=I i A[J]>P radi J=J-1
  6. Ako je I<J
  7. Ako je A[I]!=A[J]
  8. zamijeni vrijednosti elemenata A[I] i A[J]
  9. inače
  10. Ako je (I!=J) I=I+1
  11. QuickSort(A,i,I-1)
  12. QuickSort(A,I+1,j)

Primjer 4.1:

```
#include <iostream>
using namespace std;

float Pivot(float A[],int i, int j) {
	int k = j-i;
	int ind = rand()%k + i;

	return A[ind];
}

void QSort(float A[], int i, int j) {
     int I=0, J=j-I;
```

```
      float P = Pivot(A,i,j);

      while (I<J) {
            while (A[I]<P) I++;
            while (A[J]>P) J--;

            if (A[I]!=A[J]) {
               float pom = A[I];
               A[I] = A[J];
               A[J] = pom;
            }
            else if (I!=J) I++;
      }

      if (i<I-1) QSort(A,i,I-1);
      if (j>I+1) QSort(A,I+1,j);
}


void QSort(float A[],int N) {
      QSort(A,0,N-1);
}

int main () {
     int N, I;
     srand(time(0));

     do {
        cout << "N = ";
        cin >> N;
     } while (N < 2 && N > 1000);

     float *A = new float [N];

     for (int i = 0; i < N; i++) {
         cout << "A[" << i << "] = ";
         cin >> A[i];
     }

     QSort(A,N);

     for (int i = 0; i < N; i++)
         cout << A[i] << " ";

     cout << endl;

     system("pause");

     delete []A;
     return 0;
}
```

Kako niti jedan algoritam nije savršen tako i QuickSort ima svojih problema i taj se problem očituje u određivanju kružnog elementa. Naime iako je QuickSort u većini slučajeva dobar u najgorem slučaju on ima:

Uspoređivanja: N^2 Dodjeljivanja vrijednosti: $3 \cdot N^2$

Dodatno, problem je što se najgori slučaj uspostavlja kada je polje već sortirano ili obratno sortirano. Najbolji i jedini način određivanja kružnog elementa koji se danas koristi jest slučajni – odabere se slučajni indeks elementa polja i kao kružni se element uzima vrijednost koja se nalazi u tom elementu polja.

Zadaci za vježbu

- Na koji način funkcionira metoda podjeli pa vladaj?
- Zbog čega je metoda podjeli pa vladaj korisna u programiranju?
- Za što se najčešće metoda podjeli pa vladaj koristi u programiranju?
- Objasnite prednosti QuickSort u odnosu na Merge Sort i obratno.

Programski primjeri za laboratorijske vježbe

Zadatak 1.

Izradite program u kojem:

1. korisnik unosi decimalne brojeve u polje P čiji su elementi u rasponu [0, 127]. Za svaki cijeli dio broja je potrebno pronaći ekvivalentni znak u ASCII tablici. Ostvariti pretraživanje polja preko algoritma binarnog pretraživanja. Sve operacije za koje ima smisla potrebno je koristiti funkcije. Na zaslon ispisati sve potrebne podatke.

2. je potrebno u originalnom polju iz 1. točke za svaki cjelobrojni dio elemenata provjeriti dali se možda radi o Fibonaccijevom broju. Fibonaccijeve brojeve podijeliti u dvije grupe (grupe su jednako velike), elemente svake grupe je potrebno zbrojiti, s time da se zbroj prve i druge grupe još naknadno množi sa korisnikovim unosom. Nakon toga je potrebno pronaći NZM ta dva broja. Među dobivenim ostacima nastalima u potrazi za NZM omogućiti pretraživanje pomoću funkcije iz 1. točke. Sve operacije za koje ima smisla potrebno je koristiti funkcije. Na zaslon ispisati sve potrebne podatke.

3. se unose brojevi, koji mogu biti i decimalni, u jednodimenzionalno polje. Nakon unosa brojeva polje se sortira uzlazno, koristiti Merge Sort. Nakon toga je potrebno napraviti isto međutim preko vezane liste. Originalno polje mora biti sačuvano. Algoritme za pretraživanje je potrebno izvesti preko funkcija. Vrijeme koje je potrebno da se polje sortira je potrebno sačuvati. Na zaslon je potrebno ispisati sve relevantne podatke.

4. je potrebno u vezanoj listi iz 3. točke pronaći brojeve koji unese korisnik, koristiti binarno pretraživanje (koristiti funkciju iz 1. točke). Isto tako zapamtiti vrijeme pretraživanja. Nakon što pronađete broje izuzmite njegov decimalni dio, ako broj nema decimalni dio zatražite ponovni korisnikov unos, i rastavite ga na prim-faktore. Za svaki prim-faktor koji je djeljiv sa 3 ispisati ASCII znak. Za sve funkcionalnosti koje to dopuštaju i za koje ima smisla potrebno je koristiti pokazivače te funkcije. Na zaslon ispisati sve potrebne podatke.

5. se podaci koji su uneseni u 3. točci sortiraju pomoću QuickSort algoritma uzlazno te se zapamti vrijeme potrebno za sortiranje. Sada s obzirom na vremena sortiranja QuickSort te Merge Sort algoritama odredite koji je algoritam bio bolji. Isto tako nad poljem ostataka (2. točka) pronađite element koji je korisnik tražio zadnji u 2. točci sljednim pretraživanjem i zapamtite i to vrijeme pretraživanja. Odredite vremena koja su bila potrebna u oba slučaja da se pronađe potrebni element. Na zaslon ispišite sve potrebne podatke te za sve implementacije koristite pokazivače i funkcije.