Programiranje 1

Sortiranje nizova

Matej Mihelčić

Prirodoslovno-matematički fakultet
Matematički odsjek

24. siječnja 2023.

Problem sortiranja nizova

Zadan je niz od n objekata: $x_0, x_1 \ldots, x_{n-1}$, koje možemo **uspoređivati** relacijom **uređaja** $\leq$ ili $\geq$.

Kod problema **sortiranja** niza treba **preurediti** zadani niz tako da članovi budu:

- **uzlazno** ($\nearrow$) poredani: $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$ ili
- **silazno** ($\searrow$) poredani: $x_0 \geq x_1 \geq \cdots \geq x_{n-1}$

Za **preuređivanje** niza spremljenog kao **polje** koristimo **zamjene** članova niza (njihovih vrijednosti) $x_i \leftrightarrow x_j$.

U nastavku, ako nije drugačije rečeno, pretpostavljamo da niz treba **uzlazno** sortirati.

Sortiranje nizova izborom ekstrema

Ideja: koristimo **usporedbe** i **zamjene** elemenata u nizu.

- Dovedemo **najmanji** element niza $x_0, x_1, \ldots, x_{n-1}$ na **njegovo** pravo mjesto.
- To mjesto je **prvo** u cijelom nizu, pa je (nakon **zamjene**) **nova** vrijednost elementa x_0, **najmanji** element niza.
- Postupak **ponavljamo** na **skraćenom** (nesređenom) nizu $x_1, \ldots, x_{n-1}$ (duljine $n-1$).
- Niz se **skraćuje sprijeda**.
- Postupak **ponavljamo** dok ne ostane niz s **jednim** elementom (x_{n-1}). Taj niz je sigurno sortiran.

Naziv algoritma: **izbor ekstrema** $\rightarrow$ **Selection sort**.

Na **početku** algoritma imamo **nesređeni** niz. Indeks **prvog** elementa u nesređenom dijelu je 0.

Sortiranje nizova izborom ekstrema

Algoritam **uzlaznog** sortiranja izborom **ekstrema** ima **dva** glavna dijela:
Za $i = 0$, sve dok je $i < n - 1$, ponavljaj:

- U **nesređenom** dijelu niza (indeksi od i do $n - 1$) nađi **najmanji** element,
- **Najmanji** element **zamijeni** s **prvim** elementom x_i nesređenog dijela niza (ako već nije na indeksu i).

Nakon izvođenja gore navedenih koraka, **nesređeni** dio niza se **smanjio** za 1. **Prvi** element nesređenog dijela sad ima indeks $i + 1$.

Traženje **najmanjeg** elementa u **nesređenom** dijelu polja vršimo algoritmom pretraživanja (obrađeno na prethodnom predavanju).

- Inicijalizacija: **trenutno najmanji** element u nesređenom dijelu je **prvi** element. Njegov indeks je `ind_min = i`, a vrijednost `x_min =` x_i.

Sortiranje nizova izborom ekstrema

- Za elemente s indeksima $j = i + 1, \ \ldots, \ j = n - 1$ **ispitaj** je li $x_j < x\_\text{min}$.
- Ako je gornji uvjet istinit, zapamti **novu minimalnu** vrijednost $x\_\text{min} = x_j$ i **novi indeks** minimalnog elementa `ind_min = j`.

Zamjena prvog elementa u nesređenom dijelu i **minimalnog** elementa $x_i \leftrightarrow x_{\text{ind\_min}}$ se vrši korištenjem **pomoćne** varijable `temp` u **tri** koraka:

- $temp = x_i$
- $x_i = x_{\text{ind\_min}}$
- $x_{\text{ind\_min}} = temp$

Sve korake spajamo u **funkciju** za sortiranje zadanog niza.

Sortiranje izborom ekstrema - funkcija

```
1  void selection_sort(int x[], int n){
2  int i, j, ind_min, x_min, temp;
3
4  for (i = 0; i < n - 1; ++i) {
5      ind_min = i;
6      x_min = x[i];
7      for (j = i + 1; j < n; ++j) {
8          if (x[j] < x_min) {
9              ind_min = j;
10             x_min = x[j]; }
11     }
12 if (i != ind_min) { //x_min = x[ind_min]
13    temp = x[i]; //zamjena moze biti i
14    x[i] = x[ind_min];    //x[ind_min] = x[i];
15    x[ind_min] = temp; } //x[i] = x_min;
16 }
17 return; }
```

Sortiranje izborom ekstrema - primjer

Primjer: izborom ekstrema sortirajte zadano polje.

42	12	55	94	18	44	67

Sortiranje izborom ekstrema - primjer

42	12	55	94	18	44	67

i=0

x_min = x[0] = 42

ind_min = 0

Sortiranje izborom ekstrema - primjer

42	12	55	94	18	44	67

i=0

j = 1
x_min = x[1] = 12
ind_min = 1

Sortiranje izborom ekstrema - primjer

42	12	55	94	18	44	67

i=0

j = 2

x_min < x[2] = 55

ind_min = 1

Sortiranje izborom ekstrema - primjer

42	12	55	94	18	44	67

i=0

j = 3

x_min < x[3] = 94

ind_min = 1

Sortiranje izborom ekstrema - primjer

42	12	55	94	18	44	67

i=0

j = 4

x_min < x[4] = 18

ind_min = 1

Sortiranje izborom ekstrema - primjer

42	12	55	94	18	44	67

i=0

j = 5

x_min < x[5] = 44

ind_min = 1

Sortiranje izborom ekstrema - primjer

42	12	55	94	18	44	67

i=0

j = 6

x_min < x[6] = 67

ind_min = 1

Sortiranje izborom ekstrema - primjer

42	12	55	94	18	44	67

temp = x[i]
x[i] = x[ind_min]
x[ind_min] = temp

Sortiranje izborom ekstrema - primjer

12	42	55	94	18	44	67

i = 1
x_min = x[1] = 42
ind_min = 1

Sortiranje izborom ekstrema - primjer

12	42	55	94	18	44	67

i = 1

j = 2

x_min < x[2] = 55

ind_min = 1

Sortiranje izborom ekstrema - primjer

12	42	55	94	18	44	67

i = 1

j = 3

x_min < x[3] = 94

ind_min = 1

Sortiranje izborom ekstrema - primjer

12	42	55	94	18	44	67

i = 1

j = 4
x_min = x[4] = 18
ind_min = 4

Sortiranje izborom ekstrema - primjer

12	42	55	94	18	44	67

i = 1

j = 5

x_min < x[5] = 44

ind_min = 4

Sortiranje izborom ekstrema - primjer

12	42	55	94	18	44	67

i = 1

j = 6

x_min < x[6] = 67

ind_min = 4

Sortiranje izborom ekstrema - primjer

12	42	55	94	18	44	67

```
temp = x[i]
x[i] = x[ind_min]
x[ind_min] = temp
```

Sortiranje izborom ekstrema - primjer

12	18	55	94	42	44	67

i=2
x_min = x[2] = 55
ind_min = 2

Sortiranje izborom ekstrema - primjer

12	18	55	94	42	44	67

i=2

j = 3

x_min < x[3] = 94

ind_min = 2

Sortiranje izborom ekstrema - primjer

12	18	55	94	42	44	67

i=2

j = 4

x_min = x[4] = 42

ind_min = 4

Sortiranje izborom ekstrema - primjer

12	18	55	94	42	44	67

i=2

j = 5

x_min < x[5] = 44

ind_min = 4

Sortiranje izborom ekstrema - primjer

12	18	55	94	42	44	67

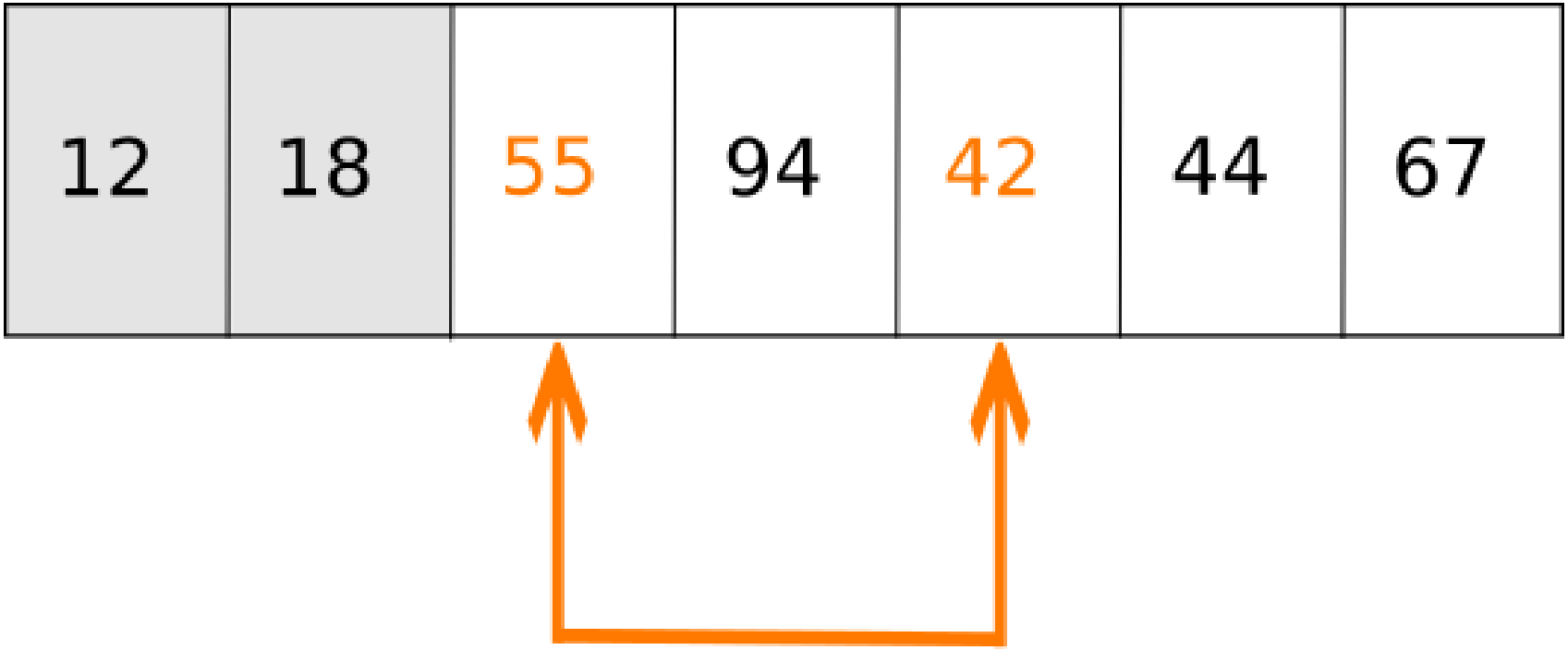
i=2

j = 6

x_min < x[6] = 67

ind_min = 4

Sortiranje izborom ekstrema - primjer




```
temp = x[i]
x[i] = x[ind_min]
x[ind_min] = temp
```

Sortiranje izborom ekstrema - primjer

12	18	42	94	55	44	67

i = 3
x_min = x[3] = 94
ind_min = 3

Sortiranje izborom ekstrema - primjer

12	18	42	94	55	44	67

i = 3

j = 4

x_min = x[4] = 55

ind_min = 4

Sortiranje izborom ekstrema - primjer

12	18	42	94	55	44	67

i = 3

j = 5

x_min = x[5] = 44

ind_min = 5

Sortiranje izborom ekstrema - primjer

12	18	42	94	55	44	67

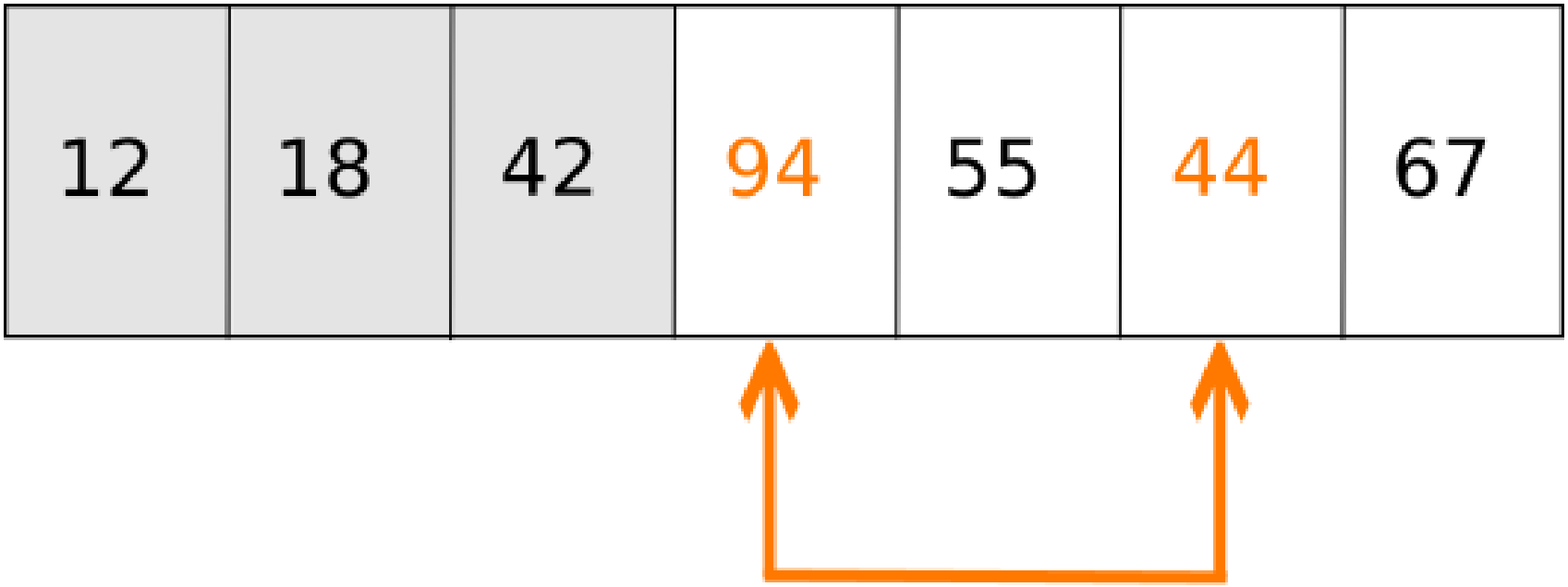
i = 3

j = 6

x_min < x[6] = 67

ind_min = 5

Sortiranje izborom ekstrema - primjer




```
temp = x[i]
x[i] = x[ind_min]
x[ind_min] = temp
```

Sortiranje izborom ekstrema - primjer

12	18	42	44	55	94	67

i = 4
x_min = x[4] = 55
ind_min = 4

Sortiranje izborom ekstrema - primjer

12	18	42	44	55	94	67

i = 4

j = 5

x_min < x[5] = 94

ind_min = 4

Sortiranje izborom ekstrema - primjer

12	18	42	44	55	94	67

i = 4

j = 6

x_min < x[6] = 67

ind_min = 4

Sortiranje izborom ekstrema - primjer

12	18	42	44	55	94	67

i = 5
x_min = x[5] = 94
ind_min = 5

Sortiranje izborom ekstrema - primjer

12	18	42	44	55	94	67

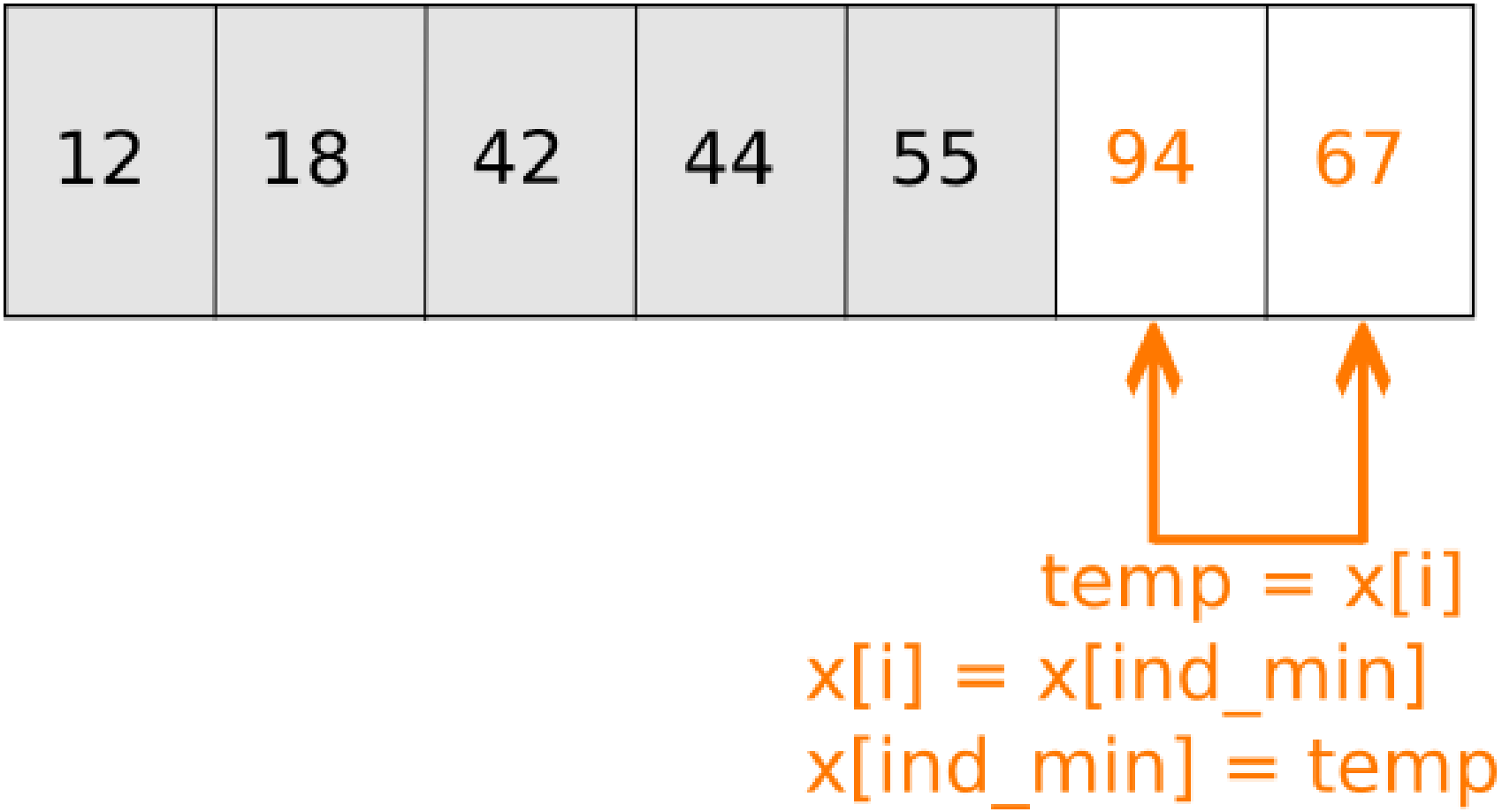
i = 5

j = 6

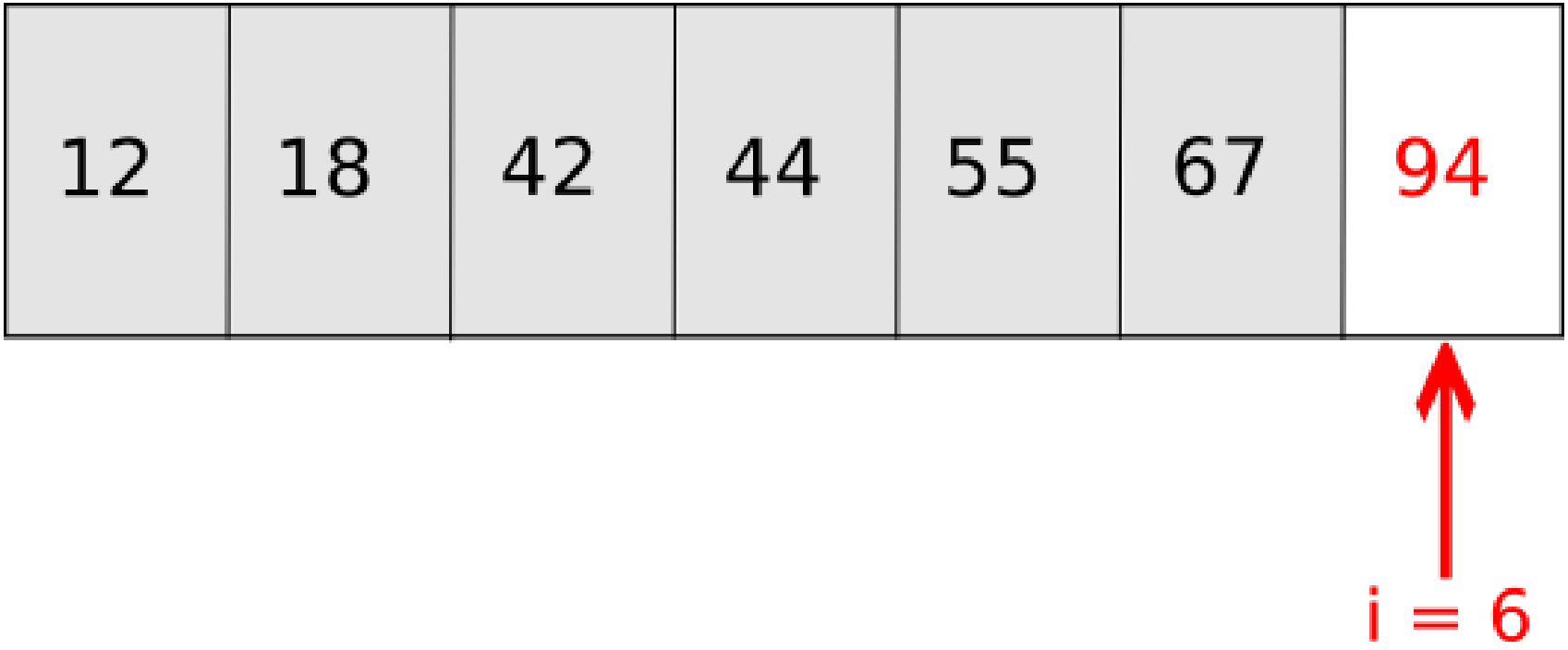
x_min = x[6] = 67

ind_min = 6

Sortiranje izborom ekstrema - primjer



Sortiranje izborom ekstrema - primjer



Sortiranje izborom ekstrema - primjer

12	18	42	44	55	67	94

Sortiranje izborom ekstrema

```
int main(void) {
    int i, n;
    int x[] = {42, 12, 55, 94, 18, 44, 67};
    n = 7;

    selection_sort(x, n);
    printf("\n Sortirano polje x:\n");
    for (i = 0; i < n; ++i) {
        printf(" x[%d] = %d\n", i, x[i]);
     }

return 0;
}
```

Sortiranje izborom ekstrema

U prethodnoj funkciji kod traženja **ekstrema** pamtimo:

- **vrijednost** ekstrema (minimuma) `x_min`,
- **indeks** elementa na kojem se ekstrem **dostiže**, `ind_min`.

Kod funkcije možemo **skratiti** tako da:

- pamtimo **indeks** elementa na kojem se ekstrem **dostiže** i **koristimo** ga kod usporedbi za **indeksiranje** članova niza.
- Trenutna **vrijednost** ekstrema je uvijek `x[ind_min]`.

Sortiranje izborom ekstrema - funkcija 2

```
1  void selection_sort(int x[], int n) {
2
3      int i, j, ind_min, temp;
4
5      for (i = 0; i < n - 1; ++i) {
6          ind_min = i;
7          for (j = i + 1; j < n; ++j)
8              if (x[j] < x[ind_min])
9          ind_min = j;
10         if (i != ind_min) {
11             temp = x[i];
12             x[i] = x[ind_min];
13             x[ind_min] = temp; }
14         }
15     return;
16     }
```

Složenost sortiranja nizova

Uspoređivati koliko je **brzo** sortiranje niza **raznim** algoritmima (u ovisnosti o duljini niza n) možemo:

- Mjerenjem **vremena izvođenja** - neprecizno (pogotovo za manje nizove).
- **Uspoređivanjem broja operacija** koje algoritam ili program obavlja.

Kod sortiranja imamo **dvije** bitno različite **elementarne operacije** (koje ne moraju jednako trajati):

- **uspoređivanje** elemenata
- **zamjena** elemenata (ili pridjeljivanje vrijednosti elementu). Svaka zamjena sadrži 3 pridjeljivanja vrijednosti elementu.

Složenost sortiranja izborom ekstrema

Kod sortiranja **izborom ekstrema** broj **usporedbi** u svakom koraku jednak je duljini trenutnog niza umanjenoj za 1. **Svaki** element osim **prvog** u trenutnom nizu uspoređuje se s trenutno **najmanjim**.

Za sve korake, **broj usporedbi** je **zbroj**:

$$(n-1)+(n-2)+\cdots+2+1=\frac{(n-1)\cdot n}{2}\approx\frac{n^2}{2}$$

Broj **usporedbi kvadratno** ovisi o n. To je generalno **loše**, postoje algoritmi koji rade znatno **manji** broj usporedbi.

U svakom koraku vrši se **najviše jedna zamjena** nekog para elemenata (ukoliko je najmanji element na **pravoj** poziciji može se dogoditi i da nema zamjene). Dakle, ukupan **broj zamjena** je najviše $n-1$. Broj **zamjena linearno** ovisi o n što je **dobro**.

Za **vremensku složenost** algoritma vrijedi:

$$T(n)\in\mathcal{O}(n^2)$$

Sortiranje izborom ekstrema - maksimum na kraju

Dosad smo **uzlazno** sortirali dovođenjem **najmanjeg** elementa na **početak**. Isti efekt možemo postići dovođenjem **najvećeg** elementa na **kraj** nesređenog dijela niza.

Ideja:

- Dovodimo **najveći** element niza $x_0, x_1, \ldots, x_{n-1}$ na **njegovo** pravo mjesto (**zadnja** pozicija u cijelom nizu).
- Postupak ponavljamo na **skraćenom** (nesređenom) nizu $x_0, \ldots, x_{n-2}$ (duljine $n-1$). Niz se **skraćuje straga**.

Indeks i vanjske petlje sadrži indeks **zadnjeg** nesređenog elementa, a petlja ide **unatrag** (od $n-1$).

Sortiranje izborom ekstrema - funkcija 3

```
1  void selection_sort(int x[], int n) {
2      int i, j, ind_max, temp;
3
4      for (i = n - 1; i > 0; --i) {
5          ind_max = i;
6          for (j = 0; j < i; ++j)
7              if (x[j] > x[ind_max])
8                  ind_max = j;
9          if (i != ind_max) {
10             temp = x[i];
11             x[i] = x[ind_max];
12             x[ind_max] = temp; }
13     }
14
15     return;
16 }
```

Sortiranje izborom ekstrema

Složenost sortiranja izborom ekstrema kod koje dovodimo najveći element na kraj je **jednaka** složenosti sortiranja izborom ekstrema kada dovodimo najmanji element na početak.

- Broj **usporedbi** je jednak:

$$\frac{(n-1)\cdot n}{2} \approx \frac{n^2}{2} \in \mathcal{O}(n^2)$$

.

- Broj **zamjena** je manji ili jednak: $n-1 \in \mathcal{O}(n)$.

Napomena: Za **silazno** sortiranje niza treba **okrenuti** ili:

- uloge **ekstrema**, **najmanji** $\leftrightarrow$ **najveći** ili (ne oboje)
- **mjesto** dovođenja ekstrema, **početak** $\leftrightarrow$ **kraj**, odnosno smjer skraćivanja, **sprijeda** $\leftrightarrow$ **straga**.

Sortiranje zamjenama susjeda - *Bubble sort*

Problem: želimo provjeriti je li zadani niz $x_0, x_1, \ldots, x_{n-1}$ već sortiran **uzlazno**.

Da bi gornje svojstvo vrijedilo, zadani niz mora biti **monotono rastući**. Za **svaki par** indeksa $j, k \in \{0, \ldots, n-1\}$ mora vrijediti $j < k \Rightarrow x_j \leq x_k$.

Algoritamski **nije potrebno** uspoređivati vrijednosti elemenata za **sve** parove indeksa (njih $\frac{n(n-1)}{2}$). **Dovoljno** je provjeriti vrijednosti samo za **sve susjedne** parove indeksa (možemo uzeti $k = j + 1$). Mora vrijediti: $x_j \leq x_{j+1}$ za **sve** $j = 0, \ldots, n-2$.

Provjera **sortiranosti** niza odgovara **predlošku** za provjeru vrijedi li neko svojstvo za sve članove niza. Povjeravamo:
$(x_0 \leq x_1) \ \wedge \ (x_1 \leq x_2) \ \wedge \ \ldots \ \wedge \ (x_{n-2} \leq x_{n-1})$.

Sortiranje zamjenama susjeda - *Bubble sort*

Moguće je konstruirati funkciju koja odgovara na zadano pitanje s **najviše** $n-1$ usporedbom:

```
int sortiran(int x[], int n)
{
    int j;
    for (j = 0; j < n - 1; ++j)
        if (x[j] > x[j + 1])
            return 0;
    return 1;
}
```

Sortiranje zamjenama susjeda - *Bubble sort*

Ako niz **nije** sortiran, **postoji** $j \in \{0, \ldots, n-2\}$ takav da je $x_j > x_{j+1}$. **Susjedni** elementi su u **pogrešnom** poretku. Funkcija za provjeru **prekida** izvođenje čim detektira **prvu** takvu poziciju.

Da bi dobili **sortirani** niz trebamo:

- **zamijeniti** poredak **susjeda** koji su u **pogrešnom** poretku (**ispraviti** njihov poredak),
- **nastaviti** provjeru kroz **cijeli** niz uz **ispravak** poretka **krivo** poredanih **susjeda**.

Primjetite da **nije moguće** sortirati opći niz u **jednom** prolazu kroz niz.

Sortiranje zamjenama susjeda - *Bubble sort*

Sortiranje **zamjenama susjeda** (eng. *Bubble sort*, *bubble* = mjehurić) bazira se na **zamjenama susjednih** elemenata u nizu.
Ideja:

- iteriramo kroz niz od **početka** do **kraja** (unaprijed $\rightarrow$),
- ako **dva susjedna** člana niza x_j i x_{j+1} **nisu** u ispravnom poretku, **zamijenimo** im pozicije (vrijednost): $x_j \leftrightarrow x_{j+1}$,
- kada stignemo do **kraja** niza (u prvom prolazu) **ponovimo** postupak.

Kada ćemo stati?

Pratite **najveći** element i **zamjene** u svakom prolazu, ako ih **nema** taj dio niza je **sortiran**.

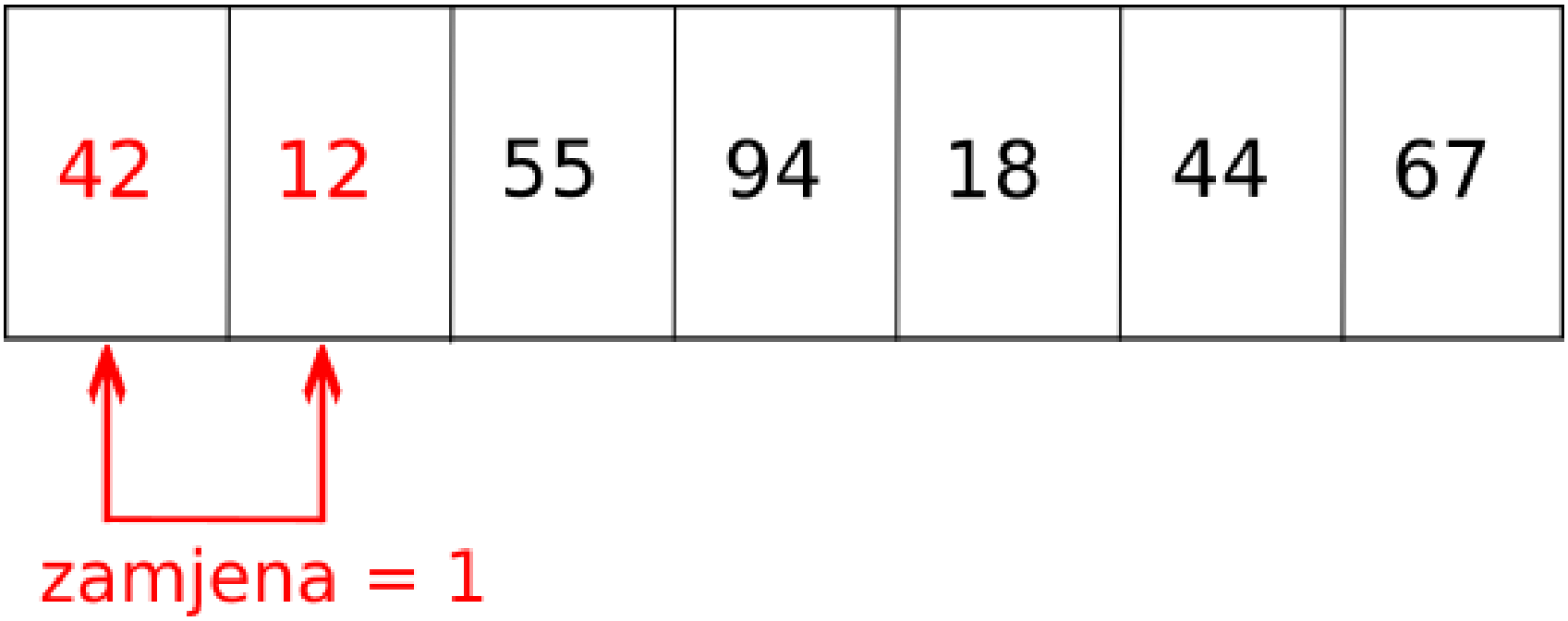
Sortiranje zamjenama susjeda - *Bubble sort*

Primjer: zamjenama susjednih elemenata sortirajte zadano polje.

42	12	55	94	18	44	67

zamjena = 0

Sortiranje zamjenama susjeda - *Bubble sort*



Sortiranje zamjenama susjeda - *Bubble sort*

12	42	55	94	18	44	67

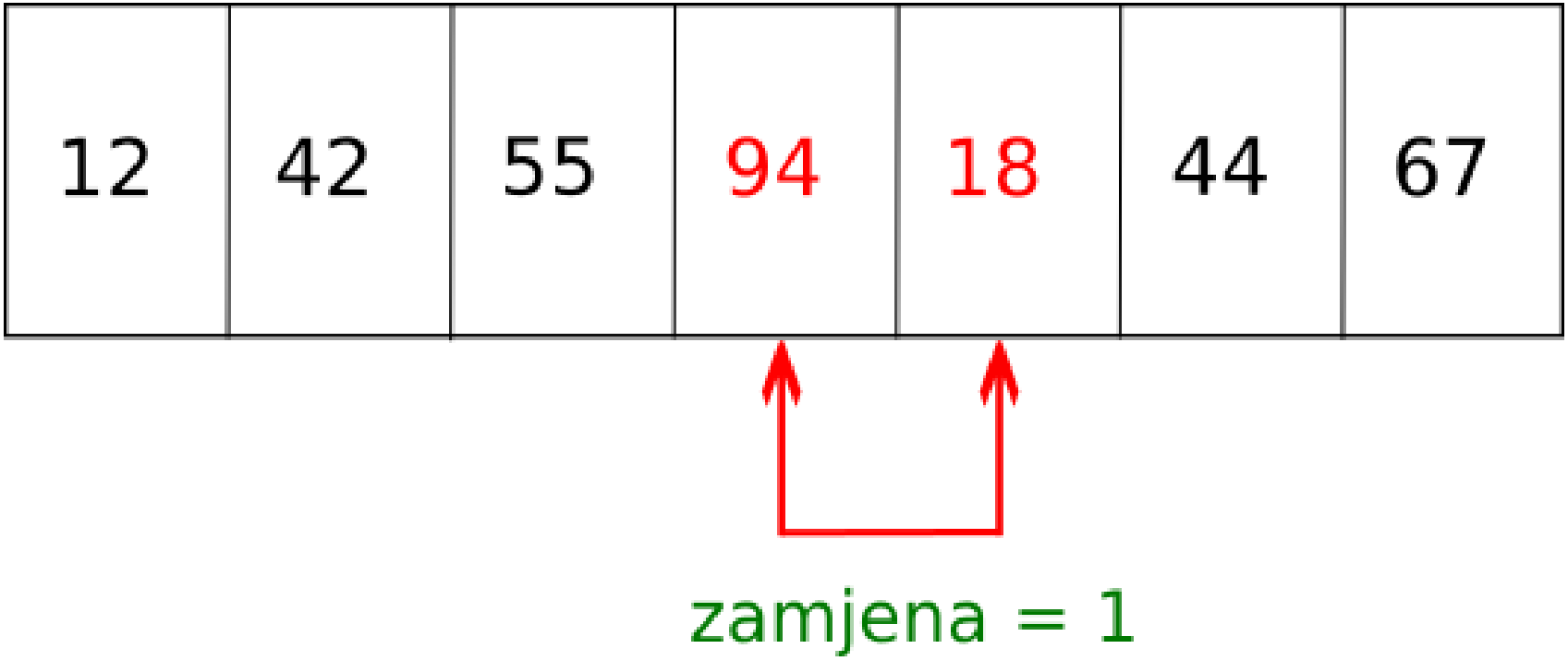
zamjena = 1

Sortiranje zamjenama susjeda - *Bubble sort*

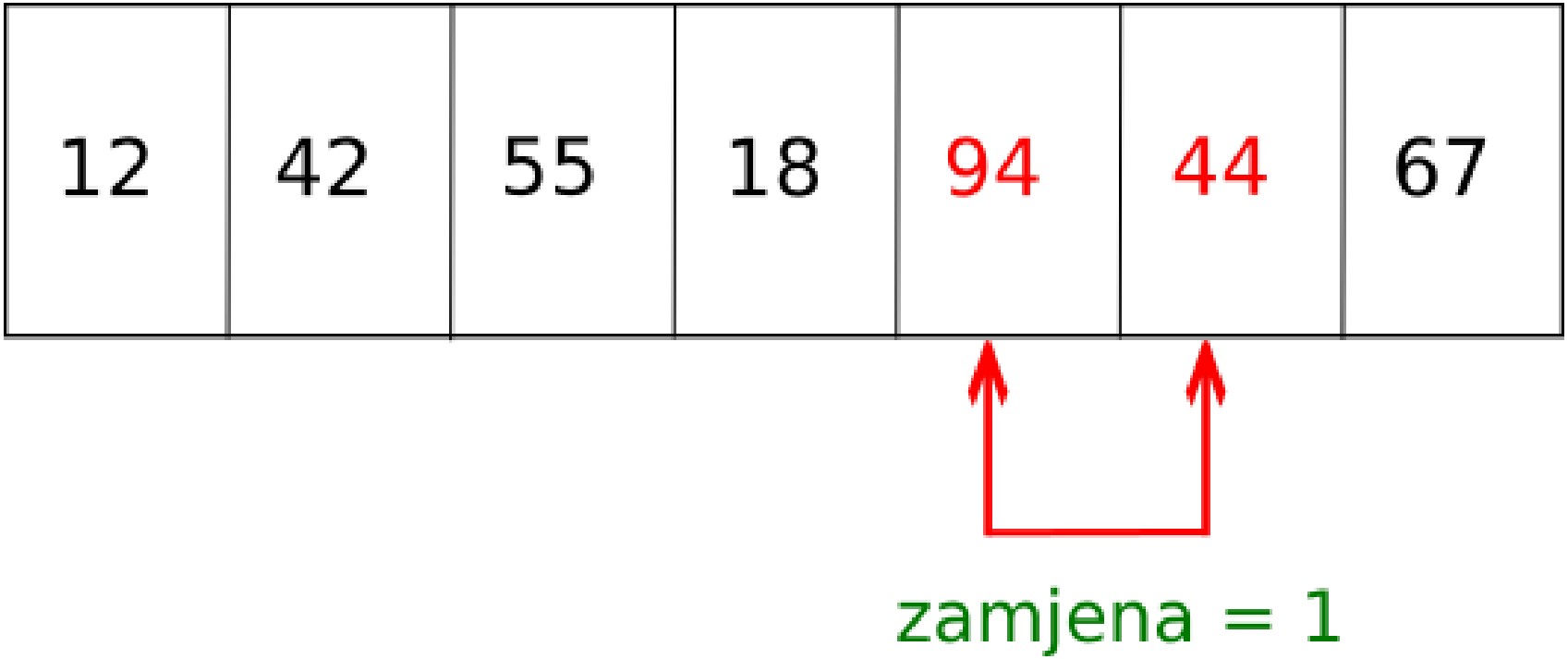
12	42	55	94	18	44	67

zamjena = 1

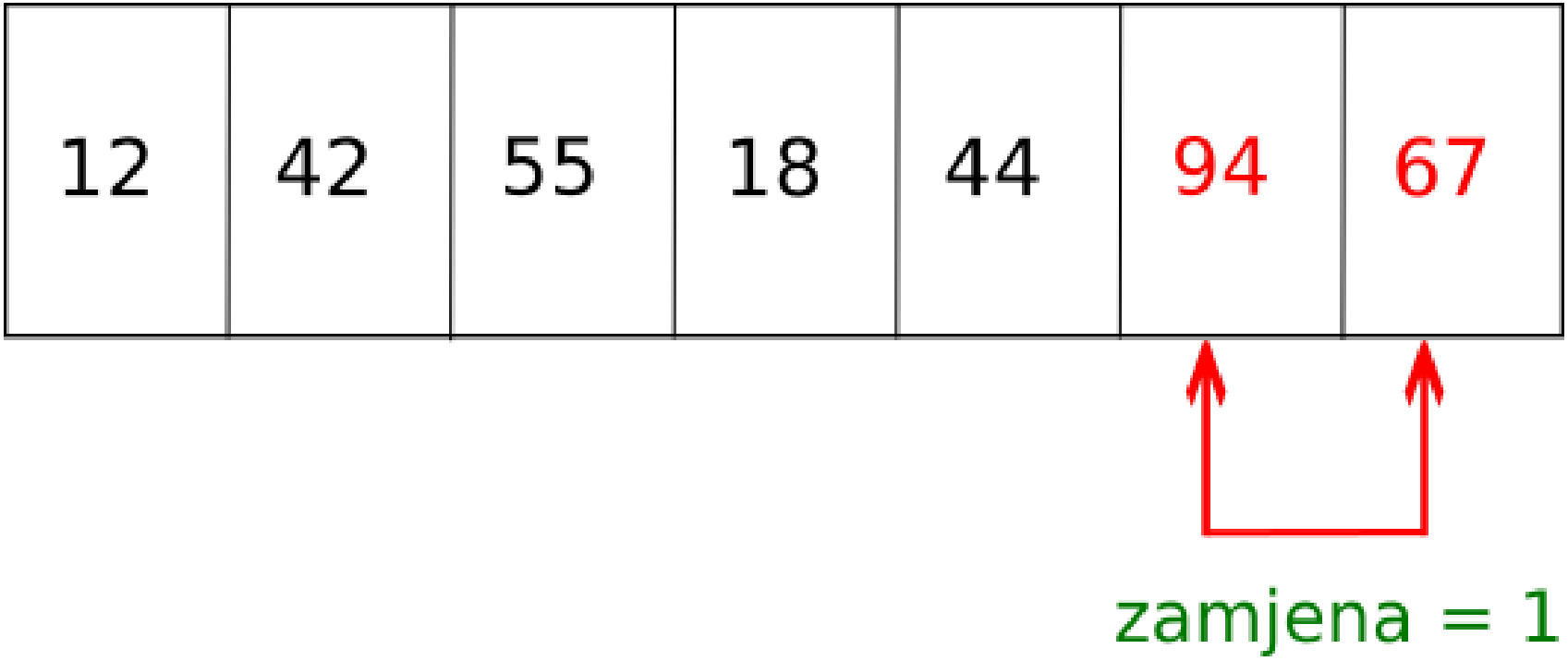
Sortiranje zamjenama susjeda - *Bubble sort*



Sortiranje zamjenama susjeda - *Bubble sort*



Sortiranje zamjenama susjeda - *Bubble sort*



Sortiranje zamjenama susjeda - *Bubble sort*

12	42	55	18	44	67	94

zamjena = 0

Sortiranje zamjenama susjeda - *Bubble sort*

12	42	55	18	44	67	94

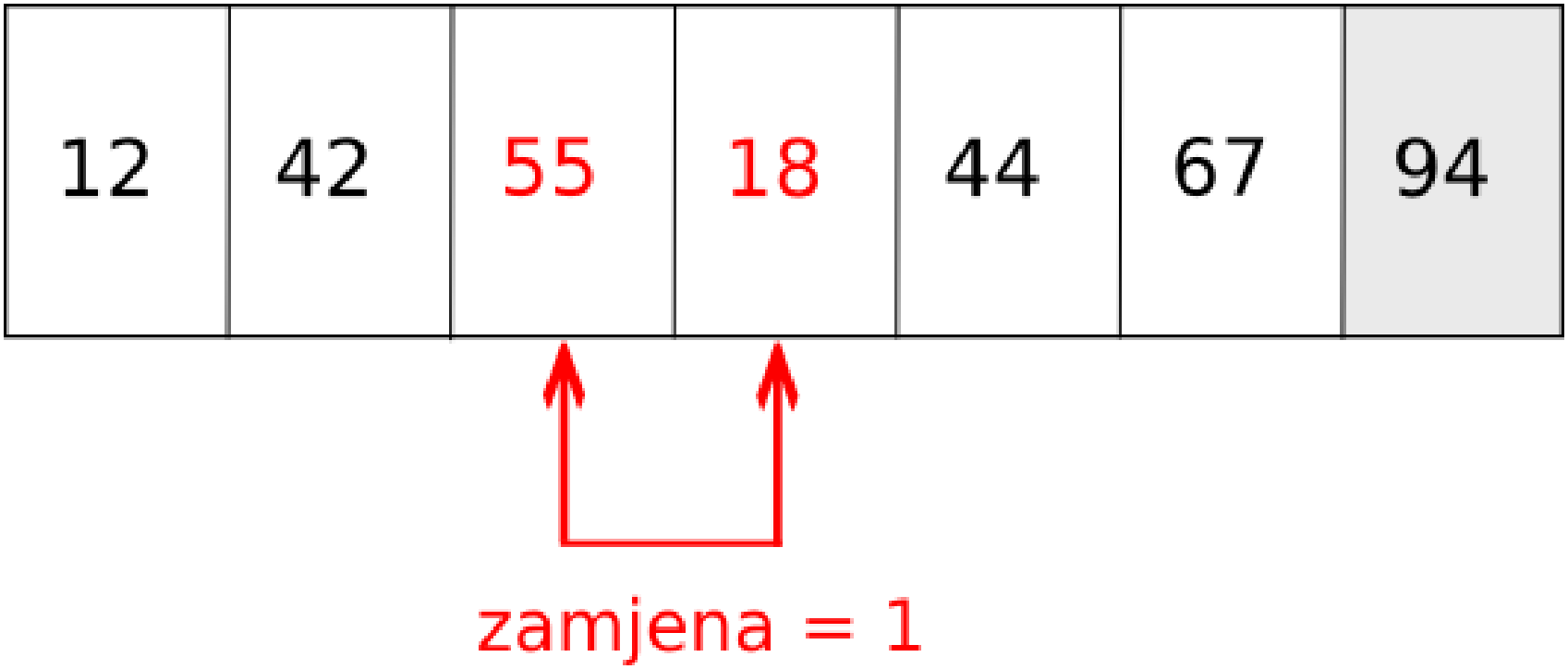
zamjena = 0

Sortiranje zamjenama susjeda - *Bubble sort*

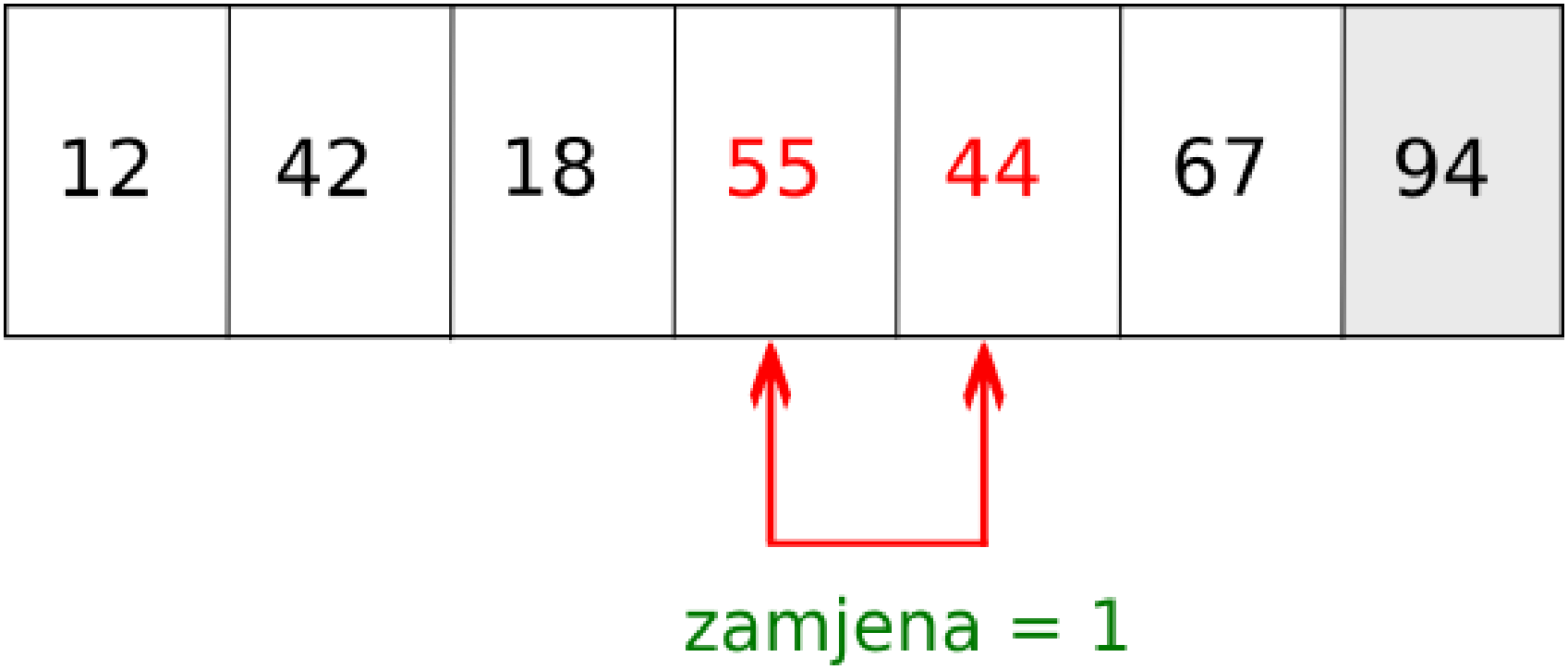
12	42	55	18	44	67	94

zamjena = 0

Sortiranje zamjenama susjeda - *Bubble sort*



Sortiranje zamjenama susjeda - *Bubble sort*



Sortiranje zamjenama susjeda - *Bubble sort*

12	42	18	44	55	67	94

zamjena = 1

Sortiranje zamjenama susjeda - *Bubble sort*

12	42	18	44	55	67	94

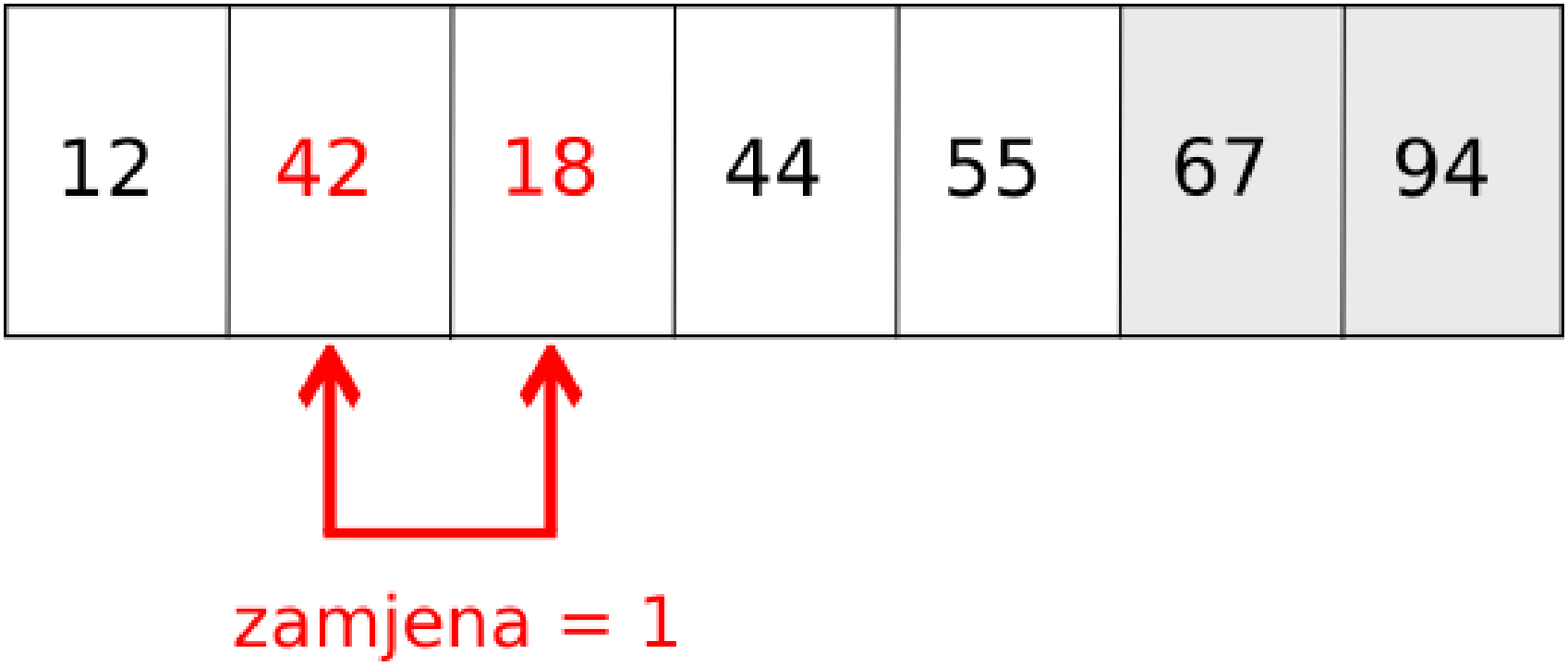
zamjena = 0

Sortiranje zamjenama susjeda - *Bubble sort*

12	42	18	44	55	67	94

zamjena = 0

Sortiranje zamjenama susjeda - *Bubble sort*



Sortiranje zamjenama susjeda - *Bubble sort*

12	18	42	44	55	67	94

zamjena = 1

Sortiranje zamjenama susjeda - *Bubble sort*

12	18	42	44	55	67	94

zamjena = 1

Sortiranje zamjenama susjeda - *Bubble sort*

12	18	42	44	55	67	94

zamjena = 0

Sortiranje zamjenama susjeda - *Bubble sort*

12	18	42	44	55	67	94

zamjena = 0

Sortiranje zamjenama susjeda - *Bubble sort*

12	18	42	44	55	67	94

zamjena = 0

Sortiranje zamjenama susjeda - *Bubble sort*

12	18	42	44	55	67	94

zamjena = 0

Sortiranje zamjenama susjeda - *Bubble sort*

12	18	42	44	55	67	94

Sortiranje zamjenama susjeda - *Bubble sort*

Kada **stajemo**?
U **prvom** prolazu **najveći** element je pozicioniran na **kraj** niza (na njegovo **pravo** mjesto). **Preostali** veći elementi se pozicioniraju prema kraju niza međutim ne moraju se pozicionirati na **pravu** poziciju.

Zaključak: nakon **prvog** koraka operacije možemo provoditi nad **skraćenim** nizom $x_0, \ldots, x_{n-2}$ (bez **posljednjeg** elementa x_{n-1}).

Niz se **skraćuje straga** kao kod sortiranja izborom ekstrema dovođenjem najvećeg elementa na kraj. U najgorem slučaju **stajemo** nakon $n-1$ prolaza na jednočlanom nizu x_0.

Sortiranje zamjenama susjeda - *Bubble sort*

```
1  void bubble_sort(int x[], int n)
2  {
3      int i, j, temp;
4
5      for (i = 1; i < n; ++i)
6          for (j = 0; j < n - i; ++j)
7              if (x[j] > x[j + 1]) {
8                  temp = x[j];
9                  x[j] = x[j + 1];
10                 x[j + 1] = temp;
11             }
12
13     return;
14 }
```

Složenost sortiranja zamjenama susjeda

U prvom prolazu **uspoređujemo** $n-1$ parova susjeda, u drugom $n-2$ i tako redom do 1 u zadnjem koraku.

Ukupan **broj usporedbi** je kao i kod **izbora ekstrema**:

$$(n-1)+(n-2)+\cdots+2+1=\frac{(n-1)\cdot n}{2}\approx\frac{n^2}{2}.$$

Broj **zamjena** može **drastično** varirati: od 0 ako je niz već **sortiran** do $\frac{(n-1)\cdot n}{2}$ (svaka usporedba prouzrokuje zamjenu) ako je niz **naopako** sortiran.

Ukupan **broj zamjena** može biti **jednak** broju usporedbi.

Složenost sortiranja zamjenama susjeda

Sažetak:

- Broj **usporedbi** je jednak

$$\frac{(n-1)\cdot n}{2} \approx \frac{n^2}{2} \in \mathcal{O}(n^2)$$

- Broj zamjena je manji ili jednak (to je **jako loše**)

$$\frac{(n-1)\cdot n}{2} \approx \frac{n^2}{2} \in \mathcal{O}(n^2)$$

Napomena: promjena **smjera** prolaza u **unatrag** ($\leftarrow$) dovodi **najmanji** na **početak**.

Za **silazno** sortiranje niza treba **okrenuti** znak **usporedbe** - **veći** ($>$) $\leftrightarrow$ **manji** ($<$).

Sortiranje zamjenama susjeda - poboljšanja

Poboljšanja algoritma:

- pamtimo logičku vrijednost, **postoji li zamjena** u **trenutnom** prolazu kroz niz. Ako **zamjena nema stajemo** (niz je **sortiran**).
- Pamtimo **indeks zadnje zamjene** u **trenutnom** prolazu. Na kraju prolaza, sve **iza** te pozicije je sortirano. U sljedećem prolazu iteriramo **samo** do te **pozicije**.

U obje varijante, za **ispravno** sortirani niz treba **samo jedan** prolaz da se ustanovi da je niz **sortiran**.

U slučaju **obratno** sortiranog niza, **nema** nikakve **uštede**. Potrebno je izvršiti svih $n - 1$ iteracija (kao i u ranijoj verziji).

Sortiranje zamjenama susjeda - f_1

```
1  void bubble_sort(int x[], int n) {
2      int i = n - 1, j, temp, zamjena;
3
4      do {
5          zamjena = 0;
6          for (j = 0; j < i; ++j)
7              if (x[j] > x[j + 1]) {
8                  temp = x[j];
9                  x[j] = x[j + 1];
10                 x[j + 1] = temp;
11                 zamjena = 1; }
12         --i; /* Smanji i za sljedeci prolaz. */
13     } while (zamjena);
14
15     return; }
```

Sortiranje zamjenama susjeda - f_2

```
1  void bubble_sort(int x[], int n) {
2        int i = n - 1, j, ind_zamj, temp;
3
4        do {
5              ind_zamj = -1;
6              for (j = 0; j < i; ++j)
7                  if (x[j] > x[j + 1]) {
8                          temp = x[j];
9                          x[j] = x[j + 1];
10                         x[j + 1] = temp;
11                         ind_zamj = j; }
12             i = ind_zamj; // i za sljedeci prolaz.
13             } while (i > 0);
14
15       return; }
```

Sortiranje zamjenama susjeda

U **Bubble sortu** uvijek krećemo od **početka** niza (s **iste** strane).

Ukoliko **jednom** krenemo od **početka**, **zatim unatrag**, opet **unaprijed** pa **unatrag**, . . . , dobit ćemo eng. *Shaker sort*. Doslovni prijevod je *streseni sort*, ime je dobio prema načinu izrade pića Vodka martini. Algoritam se još zove i *Cocktail shaker sort*, *Bidirectional bubble sort*, *Cocktail sort*, *Shuffle sort*, *Ripple sort*, *Shuttle sort*.

Slična poboljšanja kao i kod *Bubble sorta* se mogu primijeniti i na *Shaker sort*.

Sve varijante sortiranja **zamjenama susjeda** imaju **previše** zamjena stoga im je vrijeme izvođenja dosta **veliko** za **opće** nizove.

Za razliku od ove klase algoritama, sortiranje **izborom ekstrema** radi **malo** zamjena.

Sortiranje zamjenama susjeda

Bubble sort i *Shaker sort* bi potencijalno mogli raditi jako dobro za **skoro sortirane** nizove (međutim potrebna je formalna definicija izraza). Jedan način da se definira *skoro sortirani* niz je da je to niz za koji *Bubble/Shaker sort* naprave mali broj iteracija. Generalno, to je **sortirani** niz u kojeg je **slučajno** ubačen (na razne pozicije) **mali** broj **novih** članova.

Postoji **puno bolji** algoritam: **posebno** sortirati **nove članove** a onda **sortirano spojiti dva** sortirana niza. Operacija sortiranog spajanja se zove **Merge** (više na `prog 2`).

Zaključak: sortiranje **zamjenama susjeda ne treba koristiti**.

Sortiranje umetanjem - *Insertion sort*

Problem: zadan je **uzlazno** sortirani niz x s n elemenata, $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$. U taj niz treba **ubaciti** (**dodati**, **umetnuti**) zadani element `elt`, tako da **novi** niz x i dalje bude **uzlazno** sortiran.

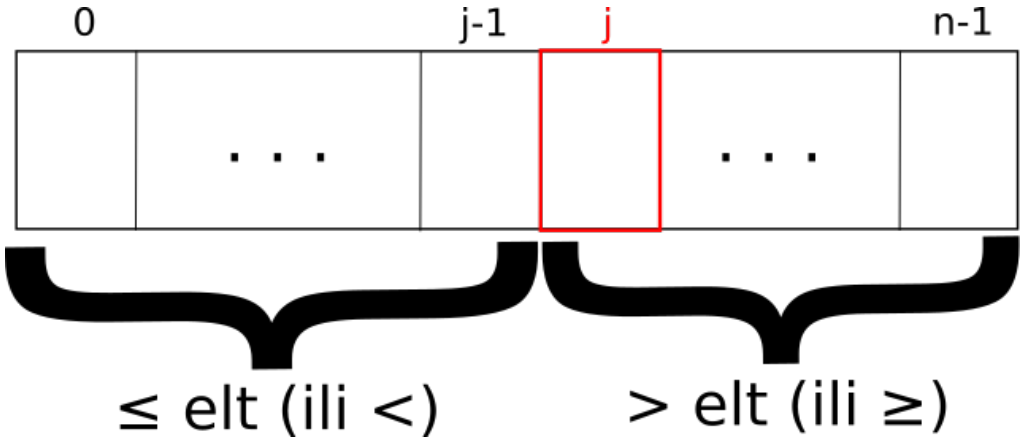
Ovaj problem se skraćeno zove **sortirano ubacivanje** elemenata u (već **sortirani**) niz.

Kad je **niz** spremljen u **polju** imamo **bitno** ograničenje: sve operacije treba napraviti u **istom** polju x.

Isti problem za **vezanu listu** ćemo obraditi na `prog 2`.

Sortiranje umetanjem - *Insertion sort*

Sortirano ubacivanje elementa `elt` u niz x ima 2 **bitna** koraka. **Prvo** treba **pronaći odgovarajuću poziciju** (budući indeks j) na koju treba ubaciti element `elt` u x.



Traženje indeksa j možemo napraviti:

- **sekvencijalno** - od **početka uzlazno** ili od **kraja silazno**,
- **binarnim** traženjem u cijelom polju.

Zatim treba **pomaknuti** blok $x[j], \ldots, x[n-1]$ za **jedno mjesto udesno** da možemo spremiti `elt` u $x[j]$.

Sortiranje umetanjem - *Insertion sort*

Uočiti da **pomaci** (kopiranja) **udesno** moraju ići **silazno** ($\leftarrow$): počevši od $x[n] = x[n-1]$ do $x[j+1] = x[j]$.

Zato se u **standardnoj** varijanti algoritma koristi **sekvencijalno** traženje od **kraja silazno** (**pomaci** se rade u istoj petlji u kojoj se vrši traženje indeksa j na koji treba ubaciti `elt`).

Ukoliko je na početku:

- `elt`$> x_{n-1}$ onda je $j = n$ i stavimo $x_n =$`elt`.
- U protivnom, kada `elt`$< x_{n-1}$, **mora** biti $j < n$. Onda treba **pronaći** j i sigurno imamo pomake **udesno**.

Kako tražimo j u **silaznoj** petlji ($\swarrow$)?

- Tražimo **najmanji** indeks j za koji je $x_j >$`elt`.
- Stajemo kada pronađemo **prvi** indeks j za koji $x_{j-1} \leq$`elt`.
- Ako **ne pronađemo** traženi j, sigurno $j = 0$.

Sortiranje umetanjem - *Insertion sort*

Krećemo od $j = n$ i u **petlji** ponavljamo sljedeće korake:

- **Pomaknemo** x_{j-1} u x_j i **smanjimo** j za 1.
- Postupak ponavljamo **sve dok** $j \geq 1$ **i** $x_{j-1} >$ `elt` (koristimo skraćeno računanje logičkih izraza).

Po završetku petlje je ili $j = 0$ ili je j **prvi** (najveći) indeks za koji je $x_{j-1} \leq$ `elt`. Dakle, pronašli smo traženu poziciju i svi pomaci su **izvršeni**.

Na kraju **umetnemo** `elt` na pravu poziciju $x_j =$`elt`.

Sortirano ubacivanje elemenata - funkcija

```
1  void ubaci_sort(int x[], int n, int elt) {
2        int j;
3
4        if (elt >= x[n - 1])
5              x[n] = elt;
6        else { /* elt < x[n - 1] */
7              j = n;
8        do {
9              x[j] = x[j - 1]; --j;
10       } while (j >= 1 && x[j - 1] > elt);
11
12       x[j] = elt;
13      }
14
15      return;
16 }
```

Sortiranje sortiranim umetanjem

Sortiranje **umetanjem** (eng. *Insertion sort*) bazira se na **sortiranom ubacivanju** sljedećeg elementa x_i u već **sortirani** početak niza $x_0, \ldots, x_{i-1}$. Početni niz od jednog elementa x_0 je već **sortiran** pa postupak ubacivanja (umetanja) ponavljamo za $i = 1, \ldots, n-1$.

Za indeks i, koraci u **jednom** prolazu kroz petlju su:

- Ako je $x_i \geq x_{i-1}$ onda je x_i već na **pravoj** poziciji i ne treba raditi ništa.
- U protivnom, za $x_i < x_{i-1}$, prvo **kopiramo** x_i u **pomoćni** element `temp`. Time oslobađamo poziciju za pomake.
- **Sortirano ubacimo** `temp` na njegovo **pravo** mjesto j_i u početni, već **sortirani** dio niza $x_0, \ldots, x_{i-1}$.

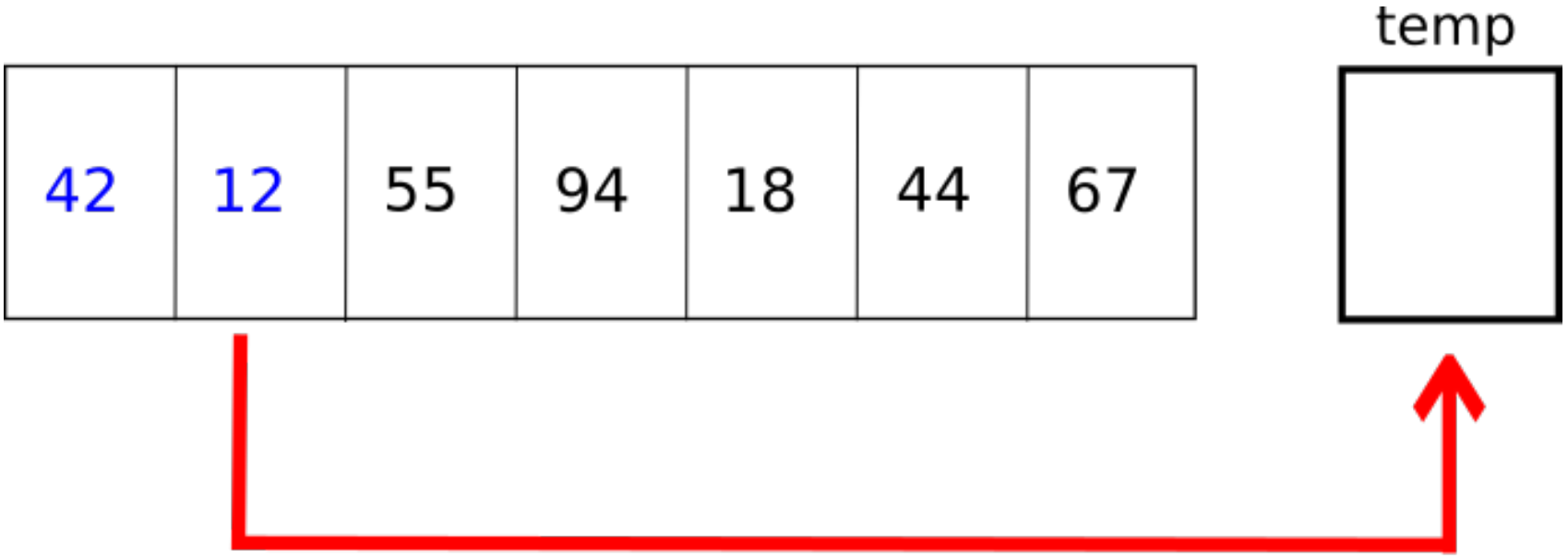
Sortiranje sortiranim ubacivanjem

Primjer: sortiranim umetanjem sortirajte zadano polje.

42	12	55	94	18	44	67

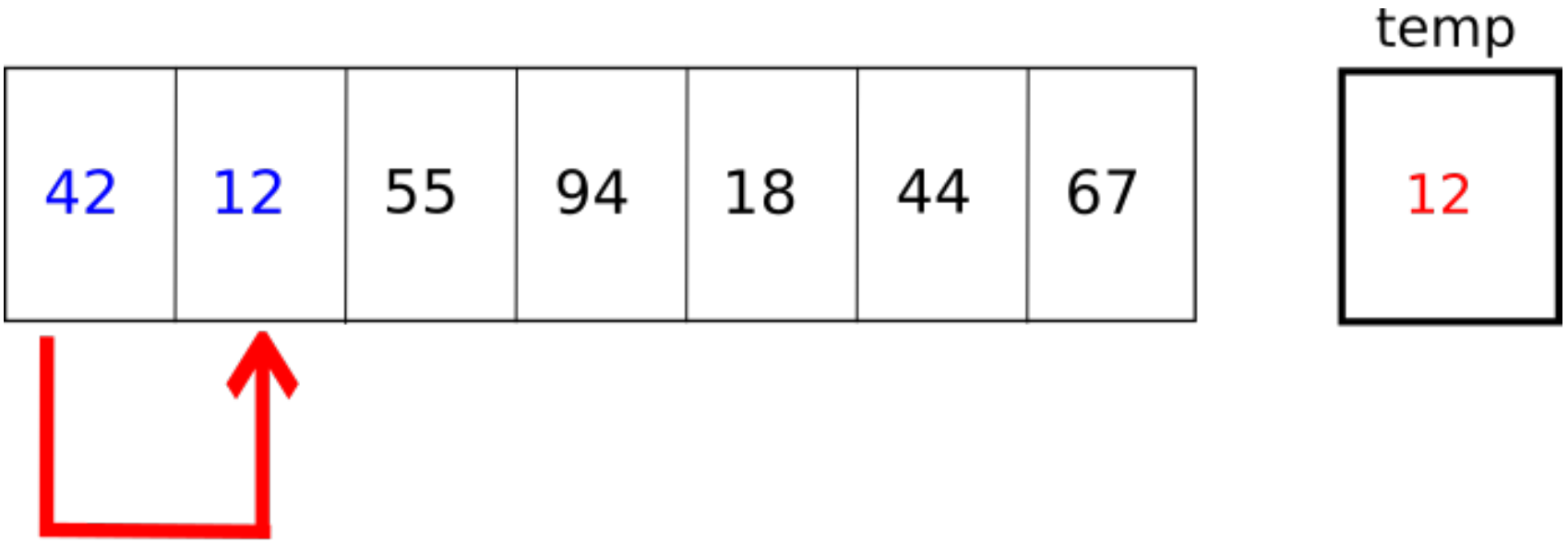
temp

Sortiranje sortiranim ubacivanjem



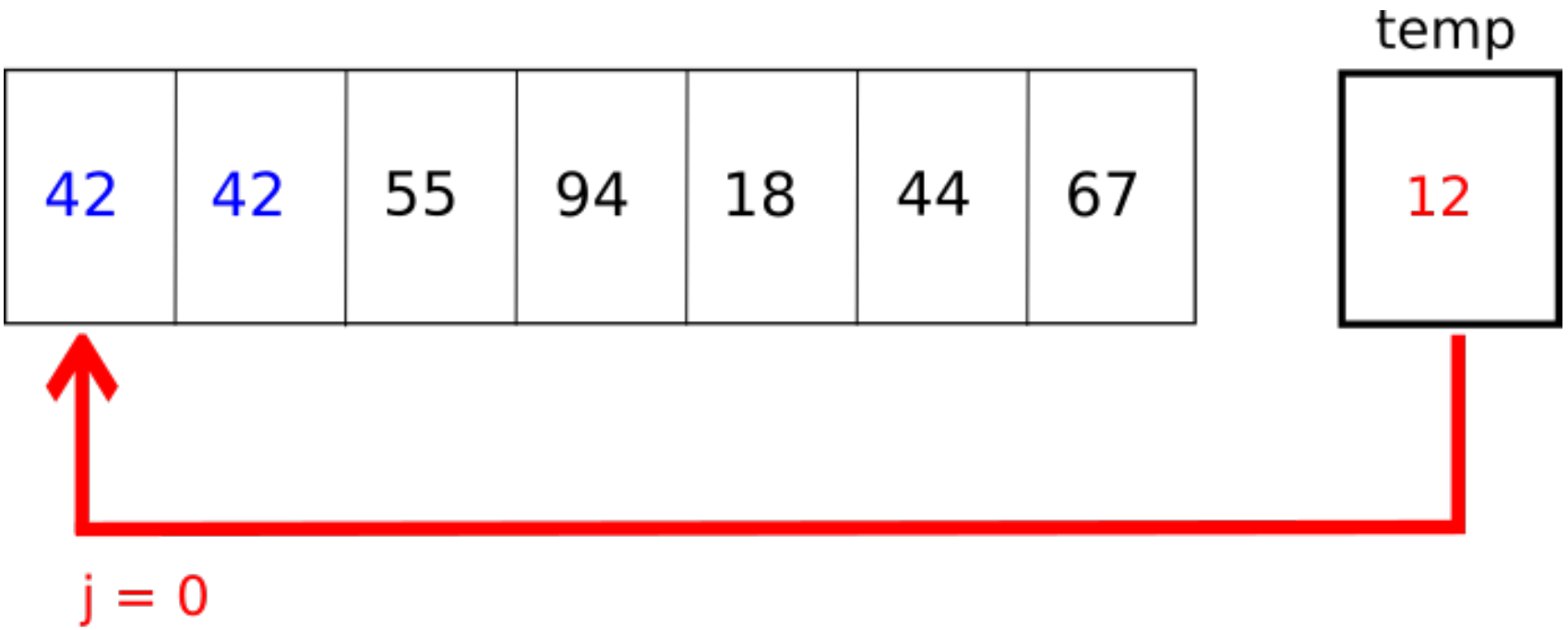
i = 1
x[1] < x[0]
temp = x[1]

Sortiranje sortiranim ubacivanjem

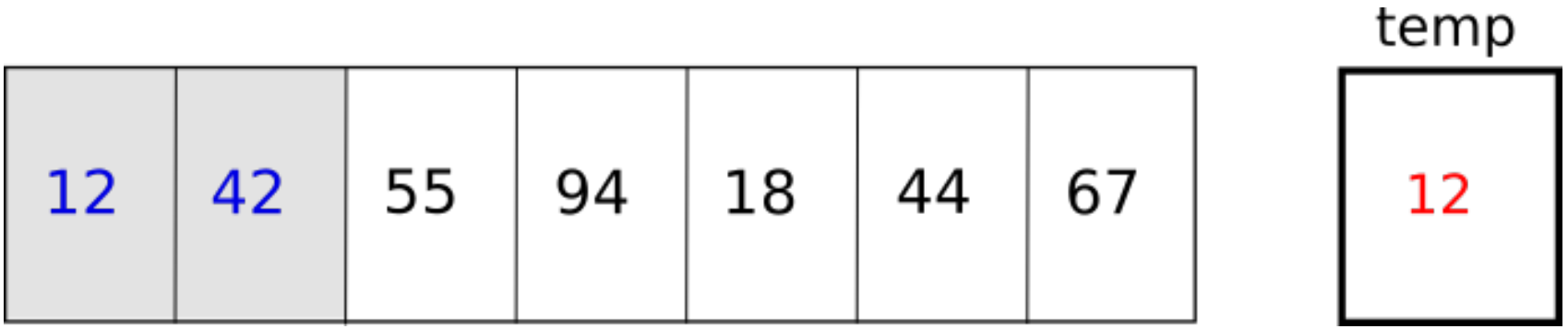


j = 1
x[0] > temp
x[1] = x[0]

Sortiranje sortiranim ubacivanjem



Sortiranje sortiranim ubacivanjem



Sortiranje sortiranim ubacivanjem

12	42	55	94	18	44	67

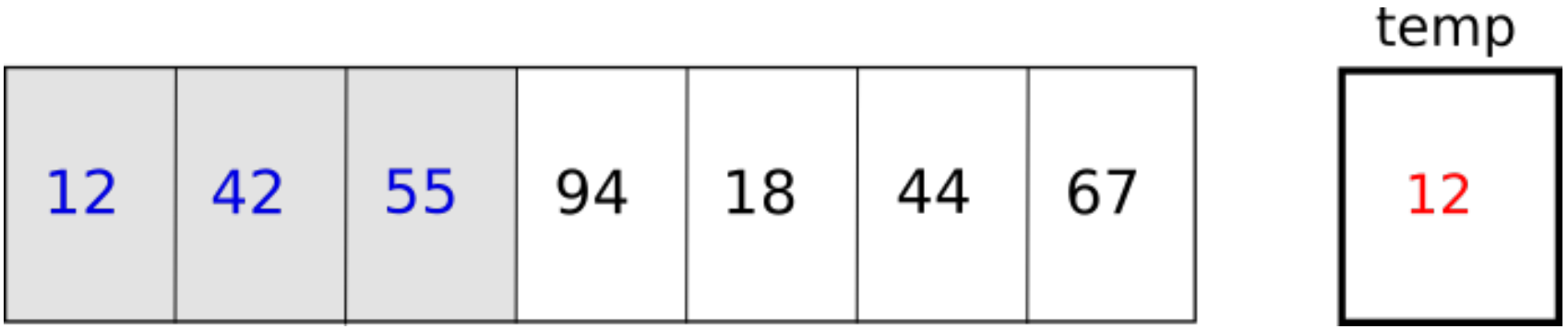
temp

12

i=2

x[2] >= x[1]

Sortiranje sortiranim ubacivanjem



Sortiranje sortiranim ubacivanjem

12	42	55	94	18	44	67

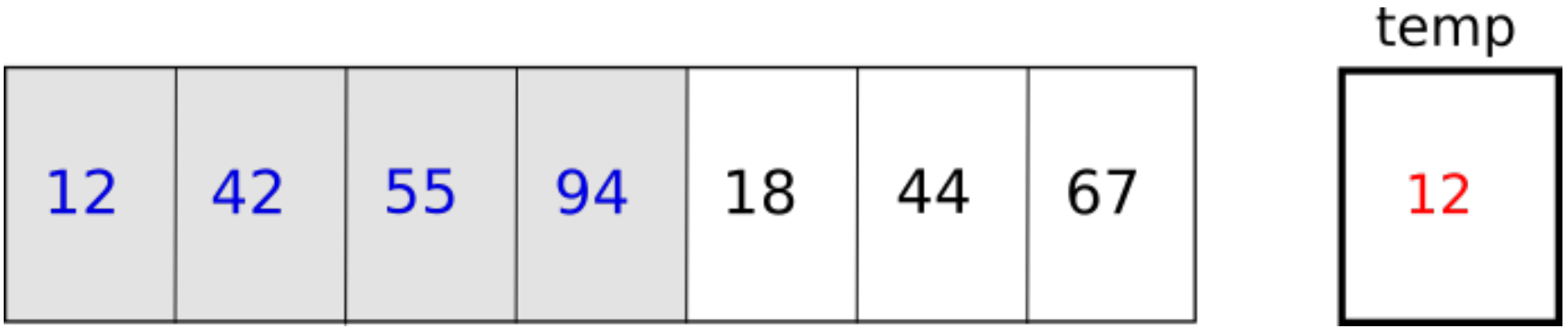
temp

12

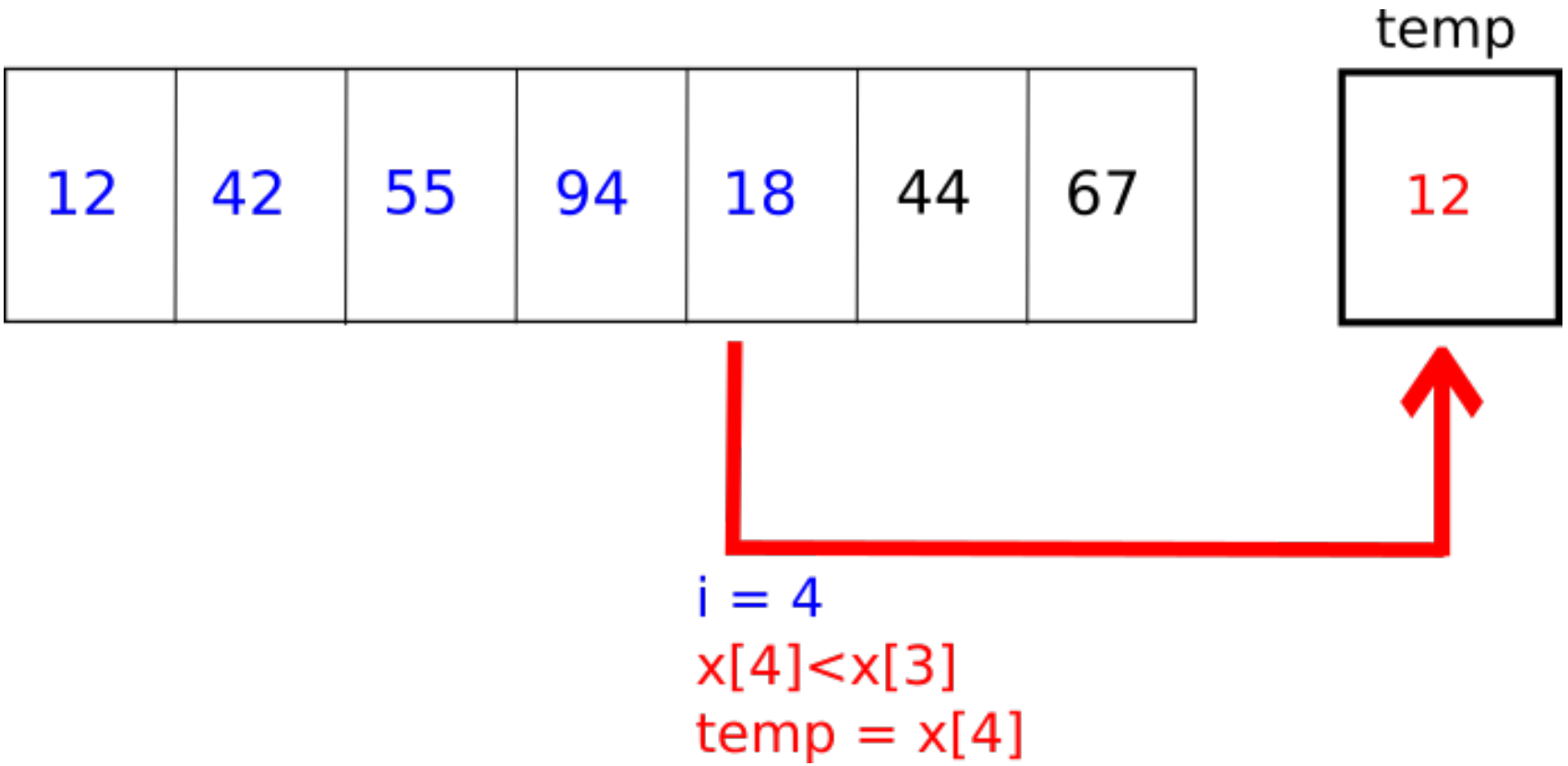
i = 3

x[3] >= x[2]

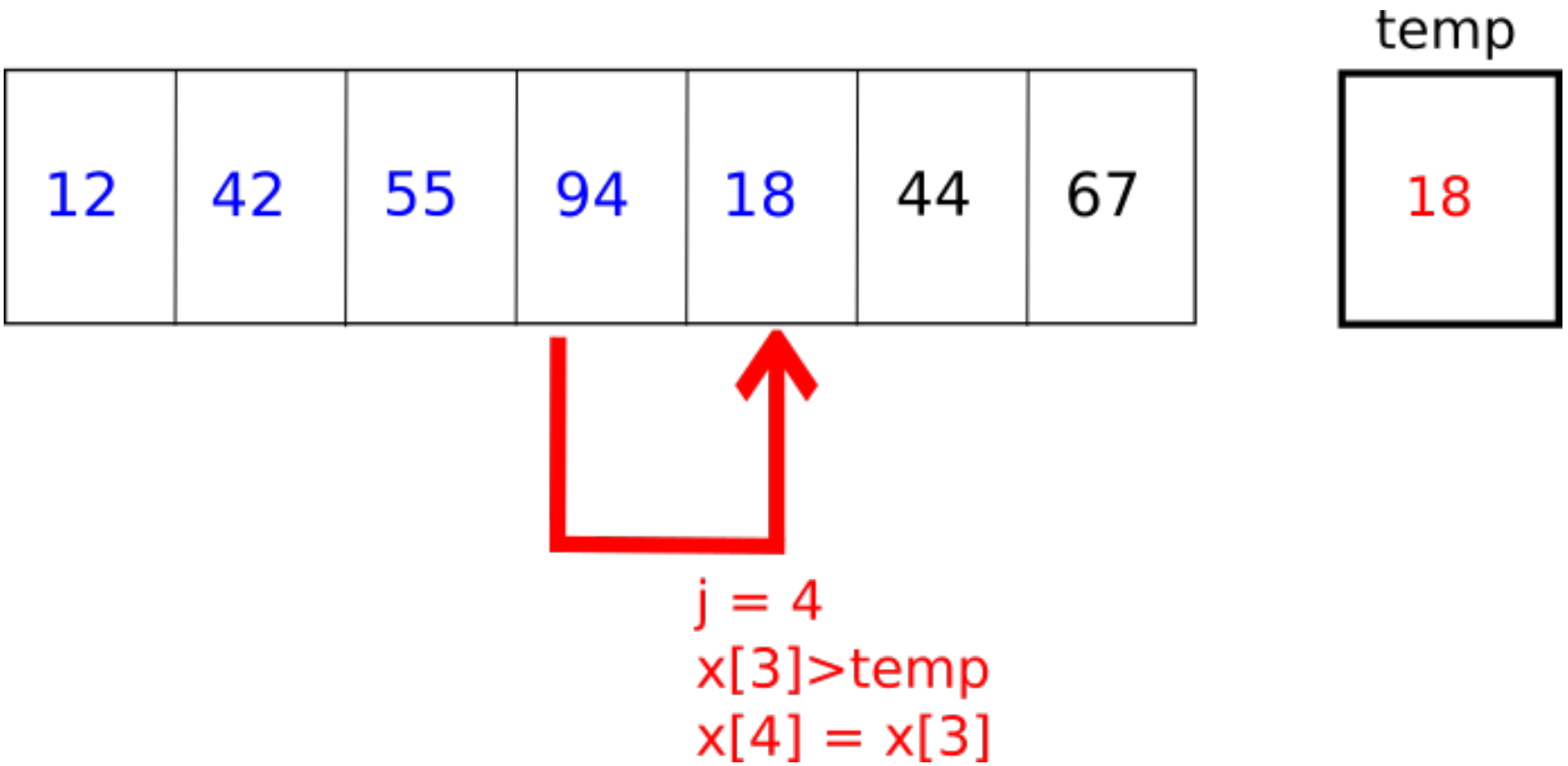
Sortiranje sortiranim ubacivanjem



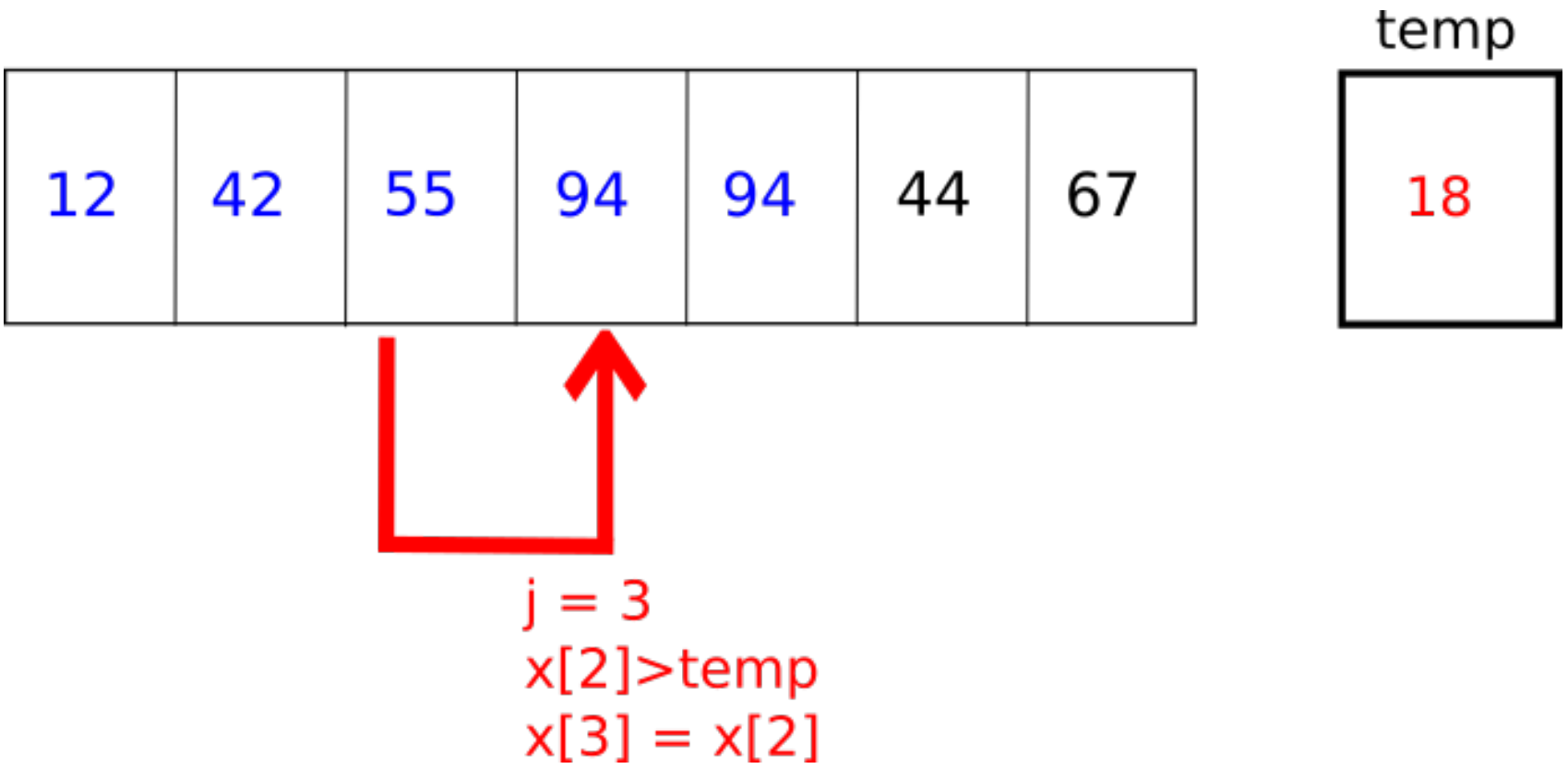
Sortiranje sortiranim ubacivanjem



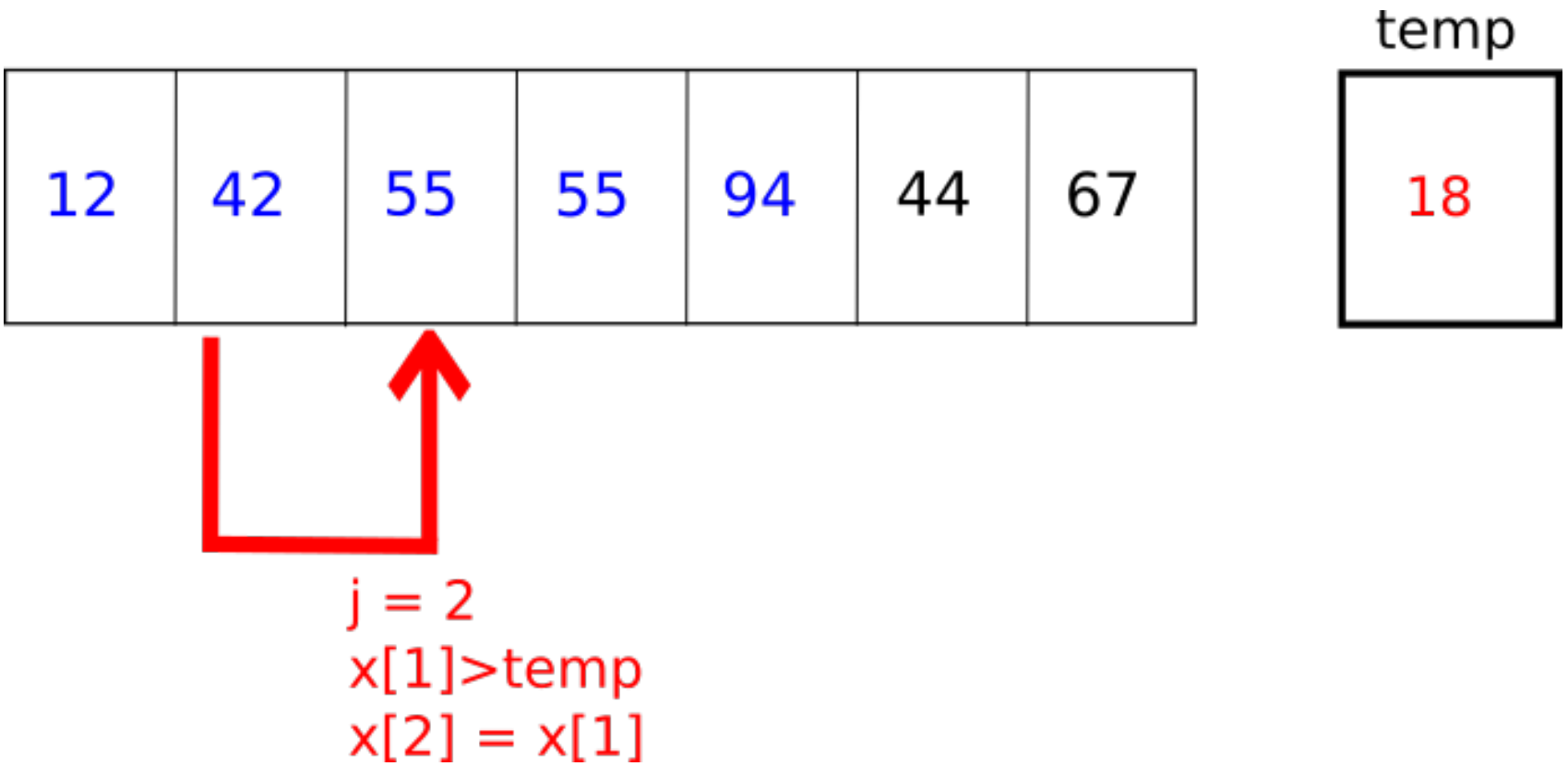
Sortiranje sortiranim ubacivanjem



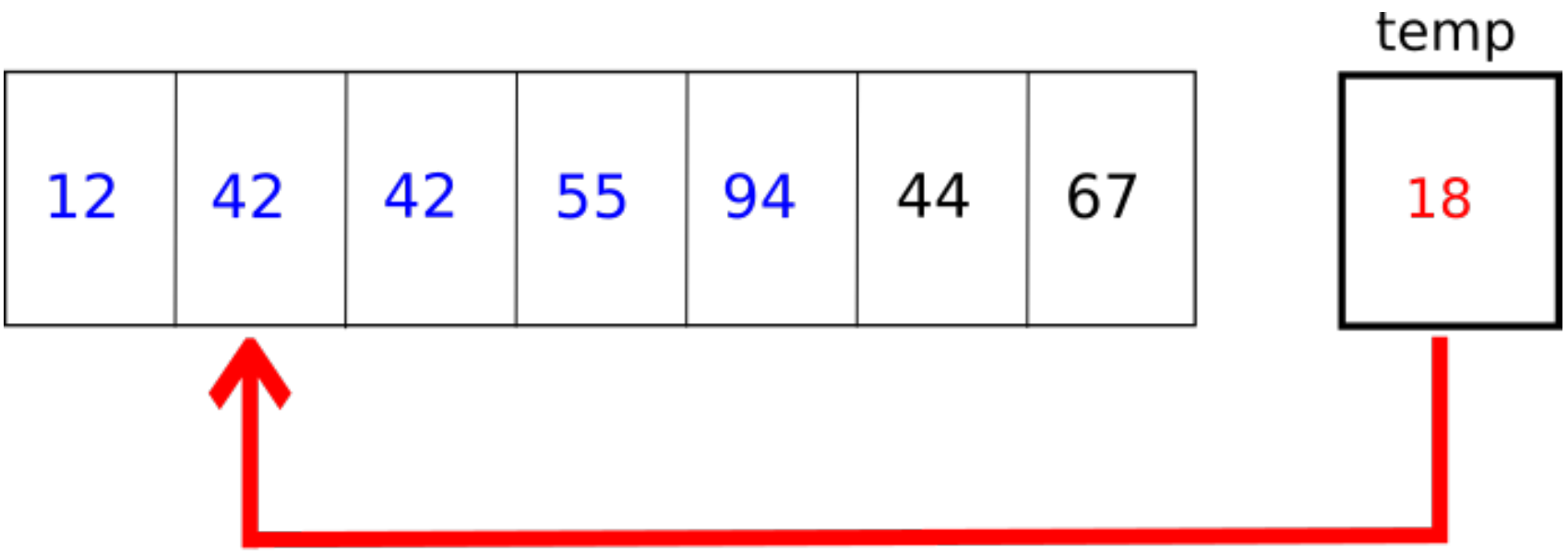
Sortiranje sortiranim ubacivanjem



Sortiranje sortiranim ubacivanjem

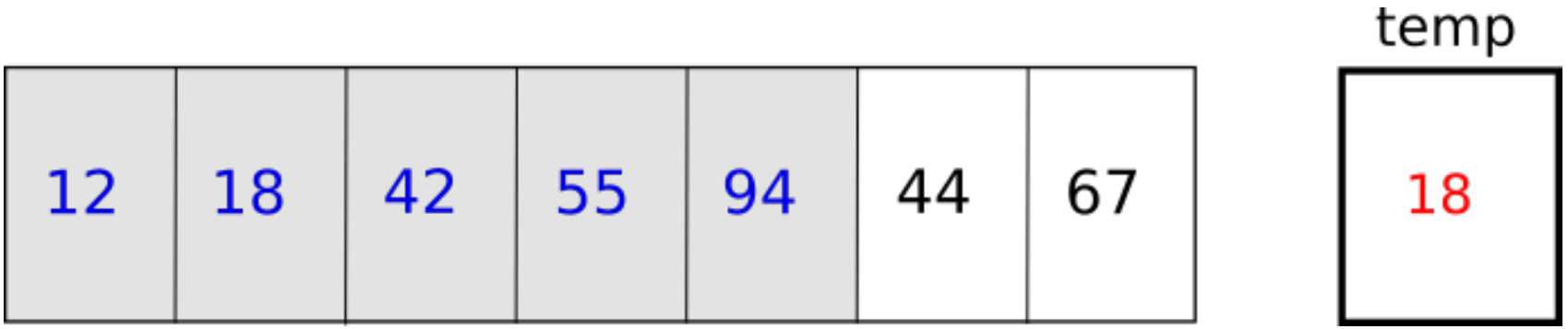


Sortiranje sortiranim ubacivanjem

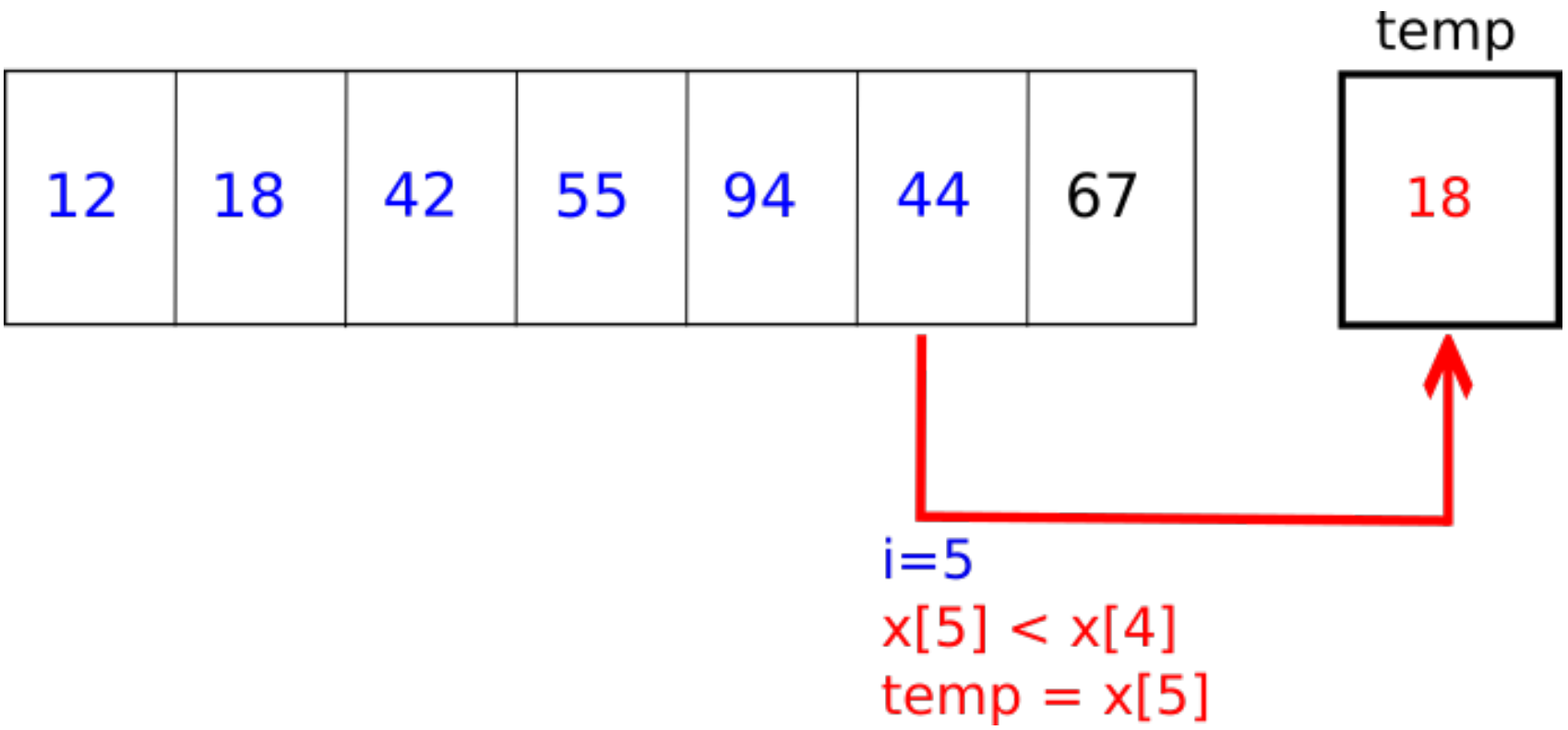


j = 1
x[0]<=temp
x[1] = temp

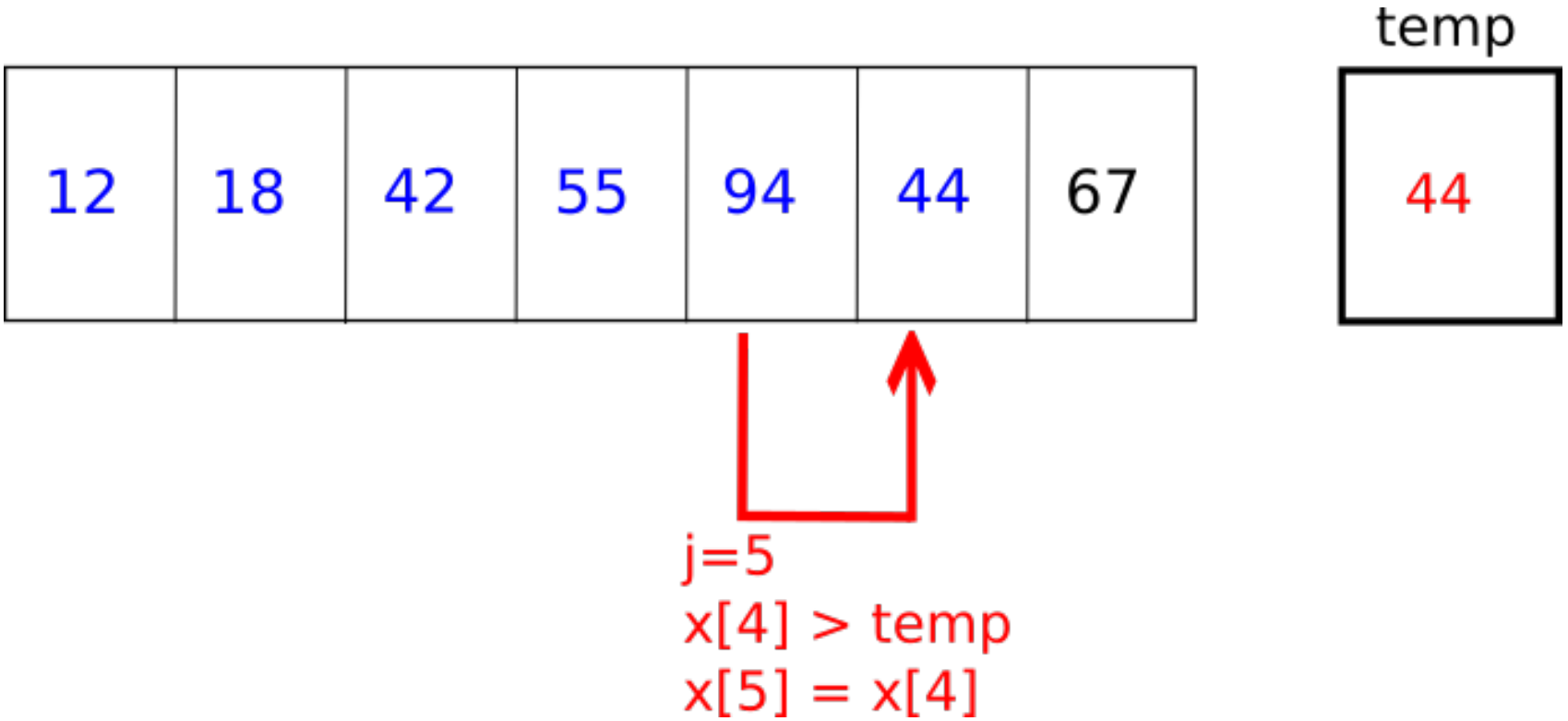
Sortiranje sortiranim ubacivanjem



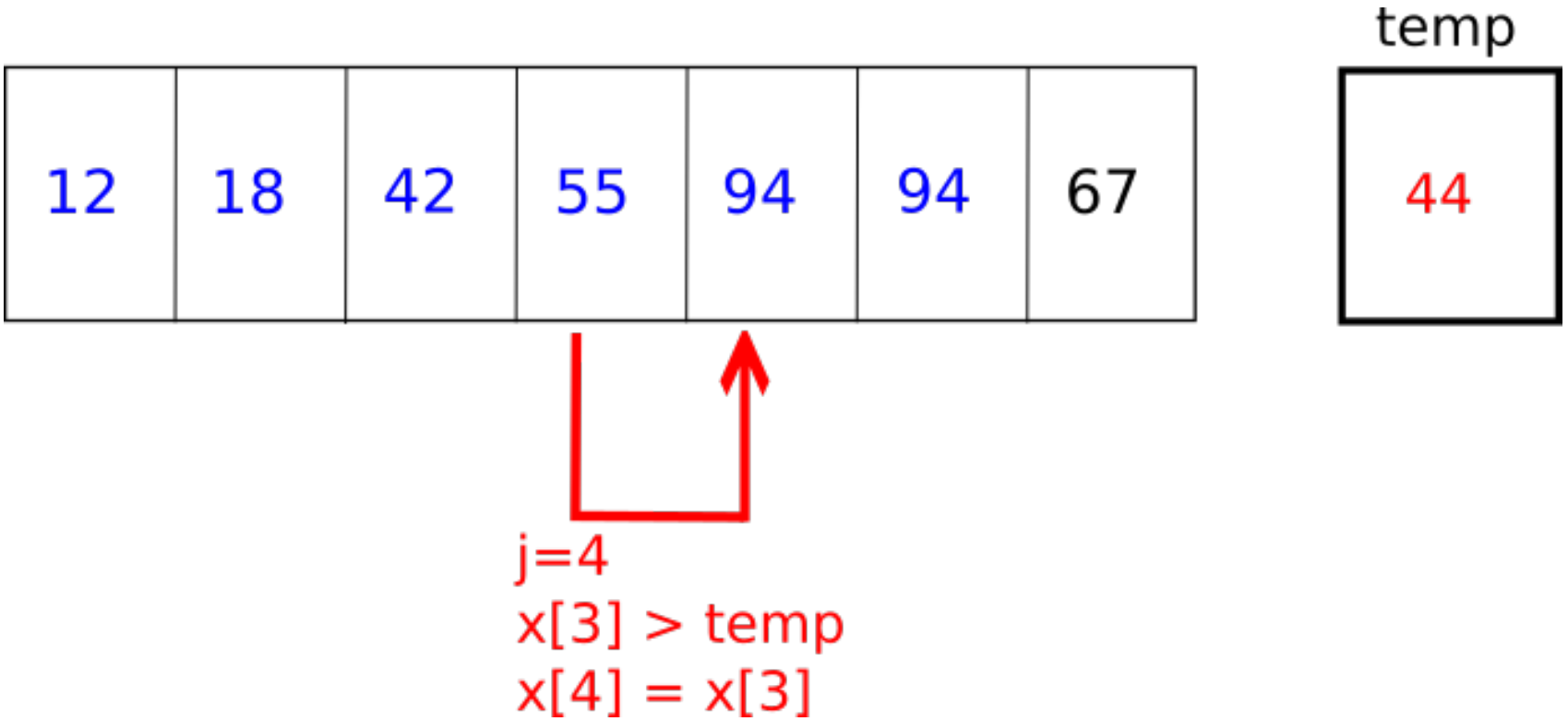
Sortiranje sortiranim ubacivanjem



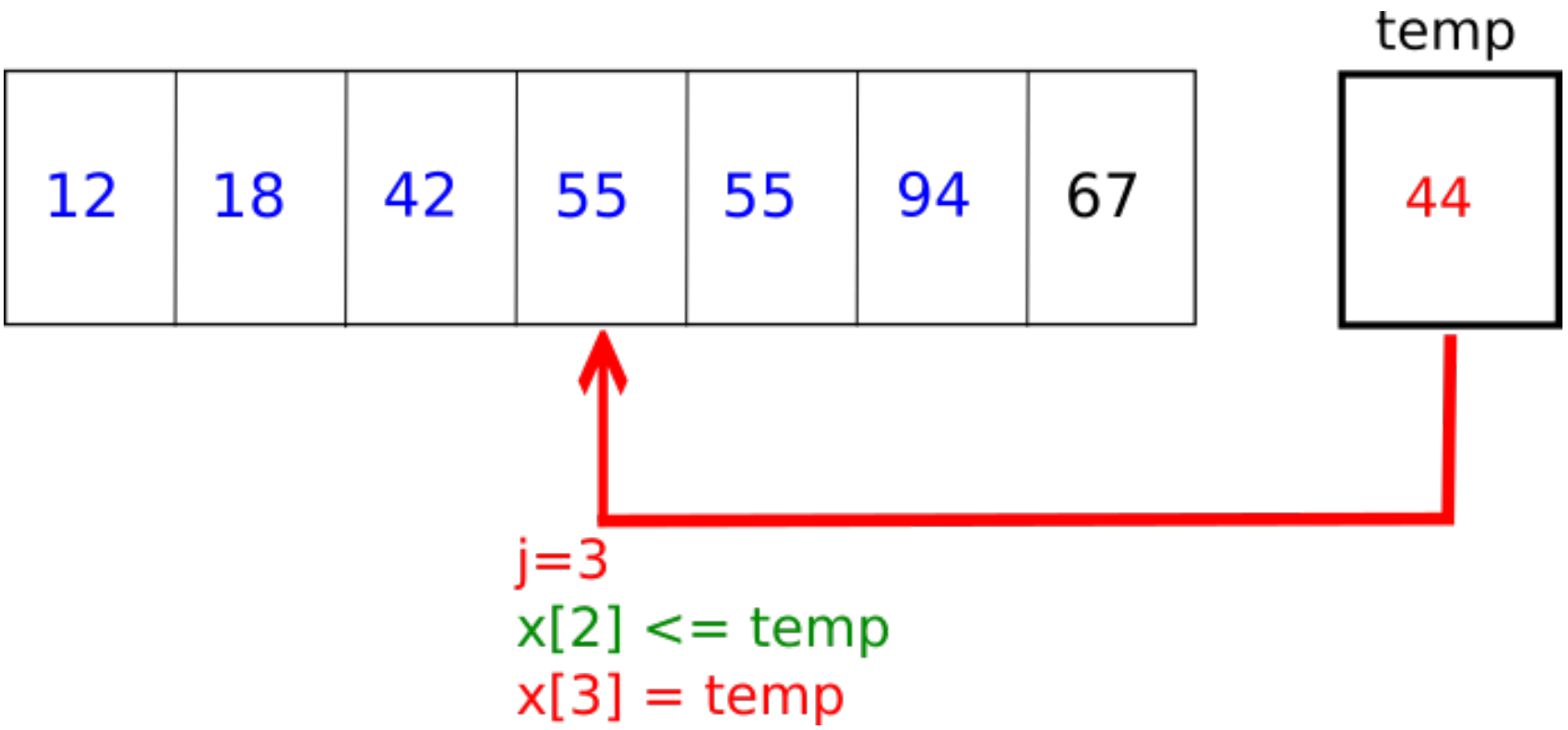
Sortiranje sortiranim ubacivanjem



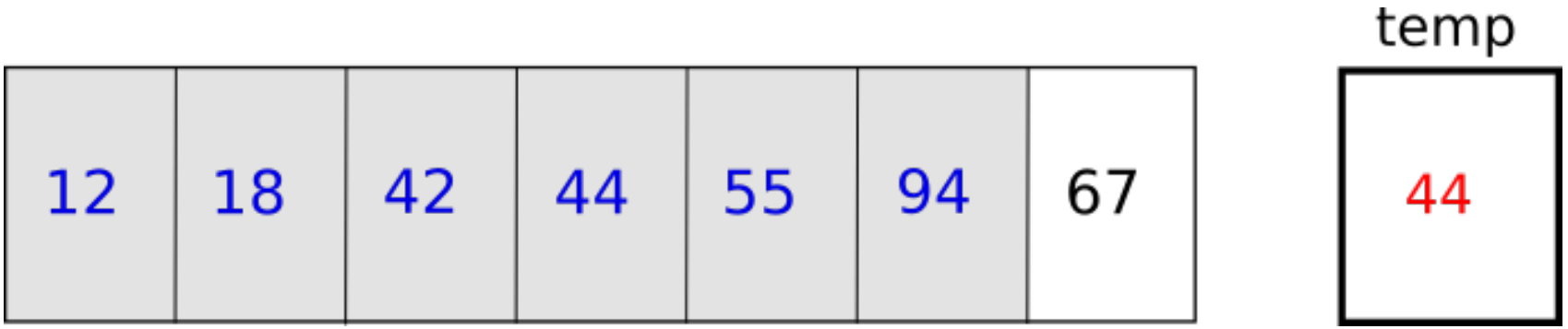
Sortiranje sortiranim ubacivanjem



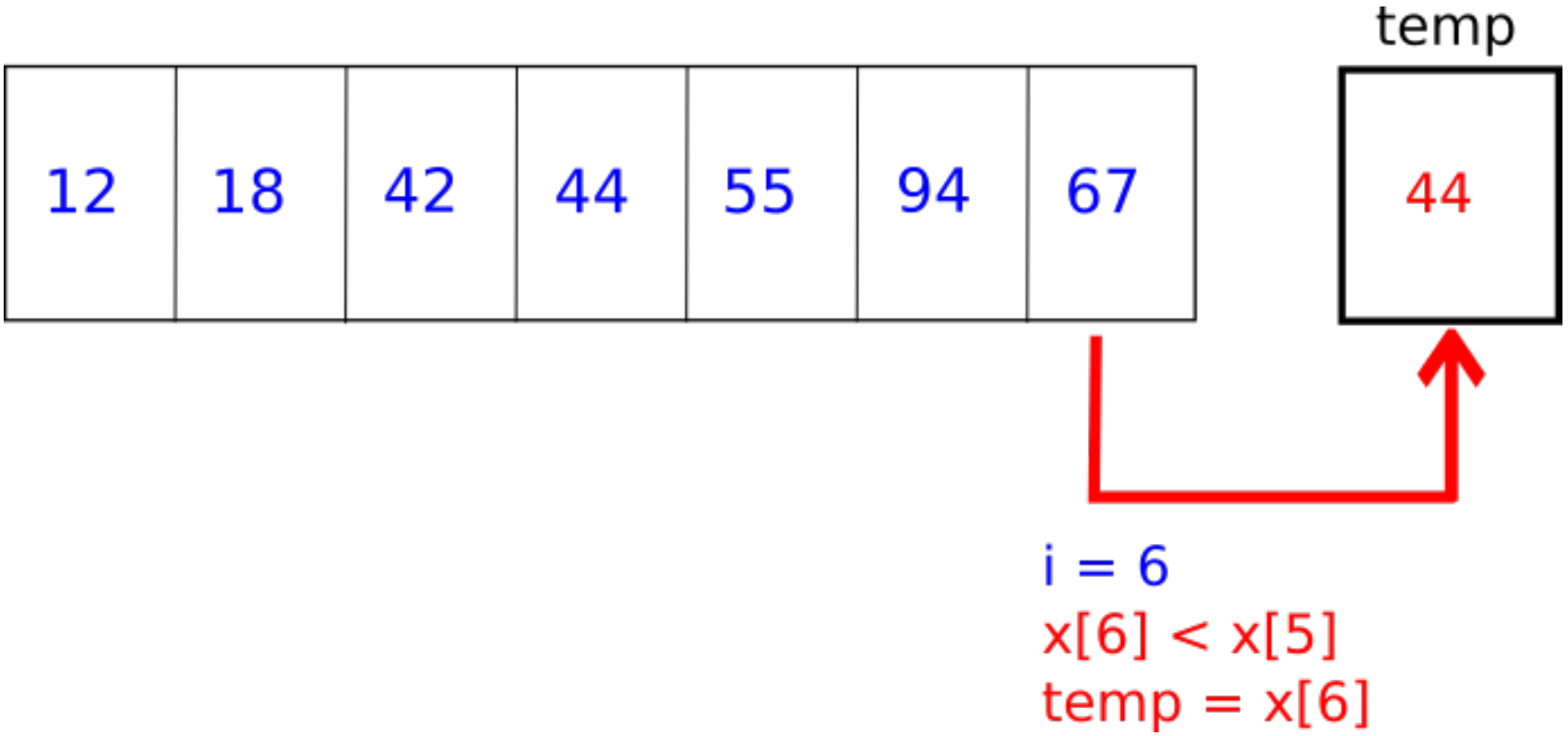
Sortiranje sortiranim ubacivanjem



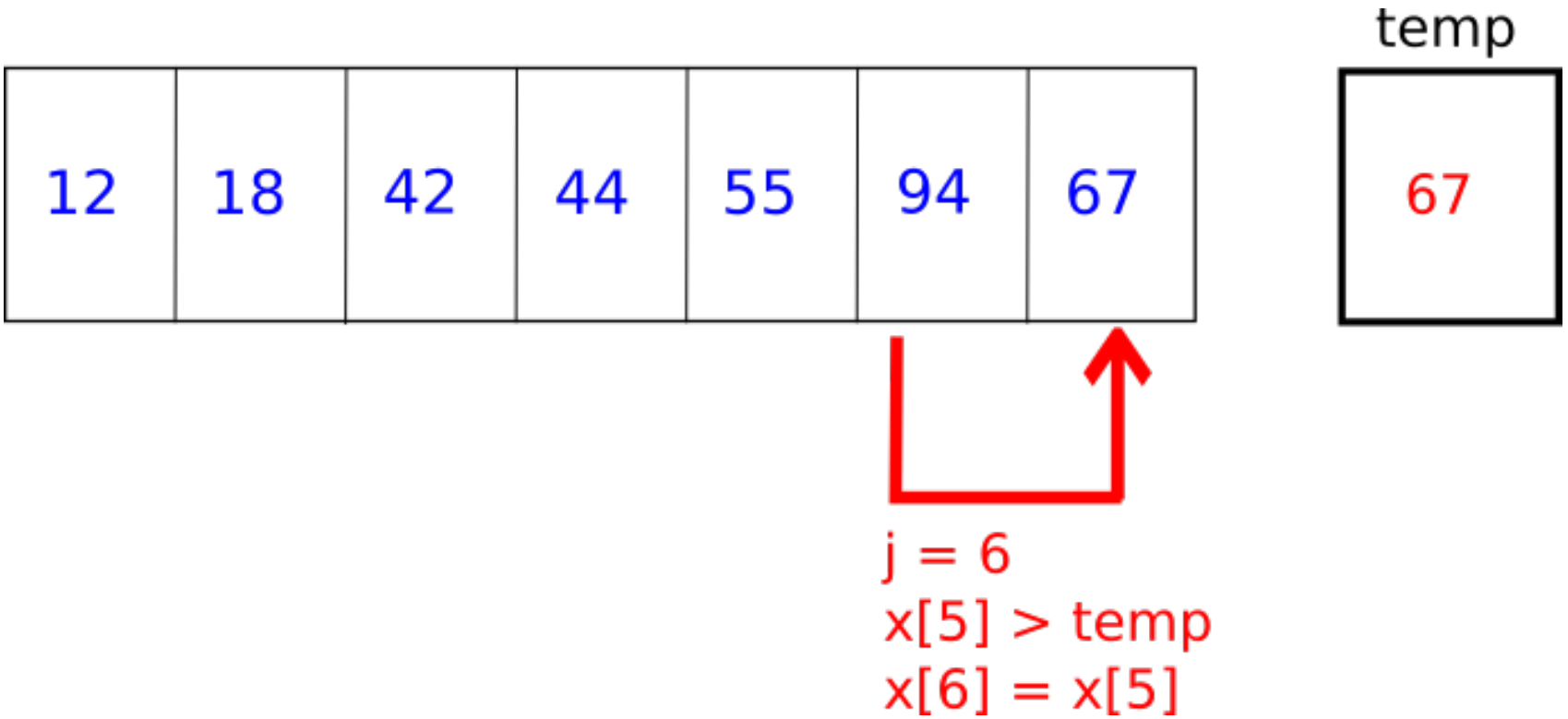
Sortiranje sortiranim ubacivanjem



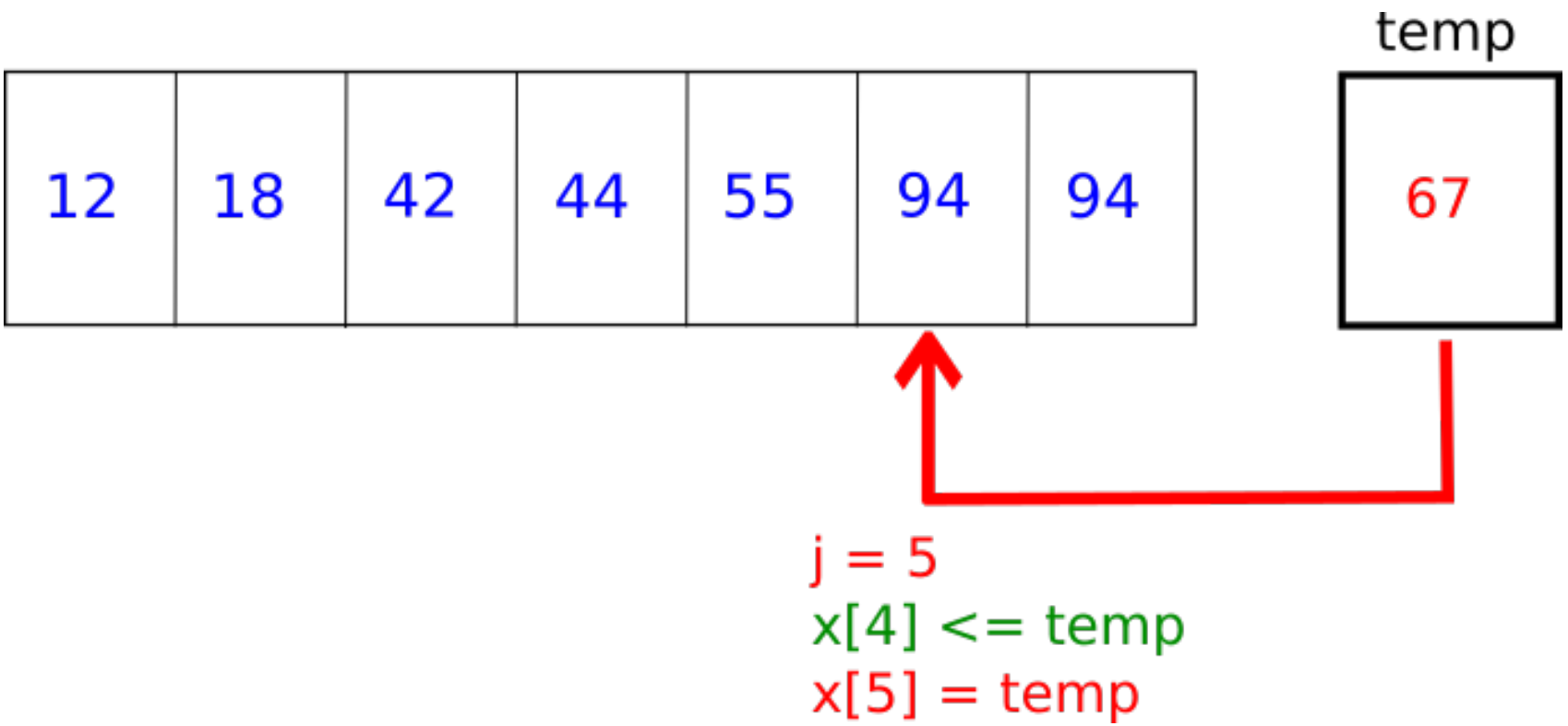
Sortiranje sortiranim ubacivanjem



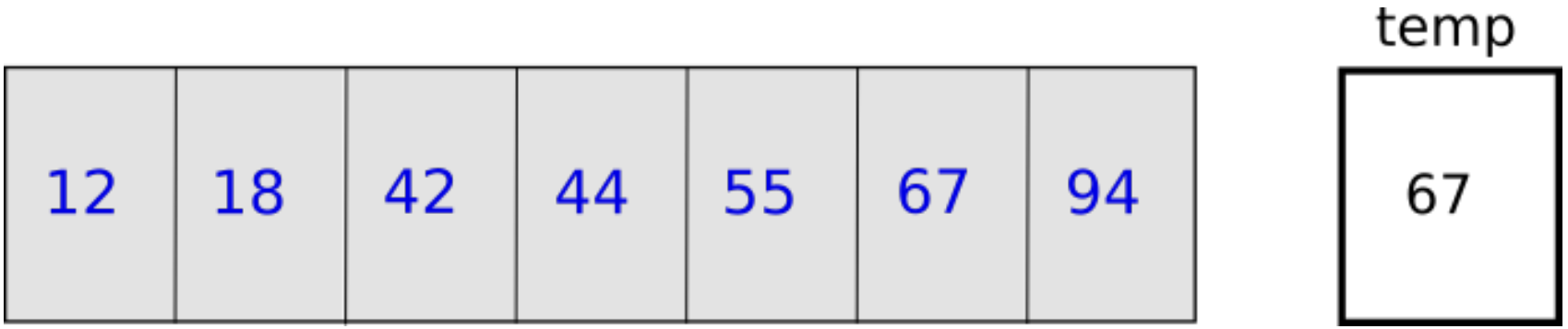
Sortiranje sortiranim ubacivanjem



Sortiranje sortiranim ubacivanjem



Sortiranje sortiranim ubacivanjem



Sortiranje umetanjem - funkcija

```
1  void insertion_sort(int x[], int n) {
2
3      int i, j, temp;
4
5      for (i = 1; i < n; ++i)
6          if (x[i] < x[i - 1]) {
7              temp = x[i];
8              j = i;
9              do {
10                 x[j] = x[j - 1]; --j;
11             } while (j >= 1 && x[j - 1] > temp);
12             x[j] = temp;
13         }
14
15     return;
16 }
```

Složenost sortiranja umetanjem

Broj **usporedbi** u i-tom koraku algoritma varira između 1 (x_i na **pravoj** poziciji, $j_i = i$) do $i - 1$ (usporedba sa svim elementima s početka niza). Najviše usporedbi se provodi ukoliko je x_i novi **najmanji** element u sređenom dijelu ($j_i = 0$).

U **najgorem** slučaju (obratno sortirani niz), ukupan **broj usporedbi** je:

$$(n-1) + (n-2) + \cdots + 2 + 1 = \frac{(n-1) \cdot n}{2} \approx \frac{n^2}{2}$$

Za već **sortirani** niz, broj usporedbi je **bitno manji** ($n - 1$).

Broj **pomaka** (dodjeljivanja) je skoro **jednak** broju **usporedbi**. Pomaci su brži od zamjena (svaka zamjena ima 3 dodjeljivanja).

Zbog toga je **Insertion sort** najbrži klasični algoritam sortiranja.

Sortiranje umetanjem - varijante i poboljšanja

U standardnoj verziji algoritma **nema** provjere `if(x[i] < x[i-1])`, već se **uvijek** kopira u `temp` i natrag.

Poboljšanja:

- **Prvo** se pronađe **najmanji** element i postavi na početak, u `x[0]`. Sada ne treba provjeravati $j \geq 1$ u petlji. **Najmanji** u `x[0]` služi kao **graničnik**.
- Za pronalaženje odgovarajuće pozicije se koristi **binarno** traženje (umjesto sekvencijalnog), a pomaci se rade **nakon** toga. Tim se **ubrzava** prvi dio algoritma.
- Kombinacija poboljšanja iz prve dvije točke.

Sortiranje umetanjem - funkcija

Standardna varijanta algoritma **Insertion sort**:

```
1  void insertion_sort(int x[], int n) {
2
3    int i, j, temp;
4
5    for (i = 1; i < n; ++i) {
6        temp = x[i];
7        for (j = i; j >= 1 && x[j - 1] > temp; --j)
8            x[j] = x[j - 1];
9         x[j] = temp;
10      }
11
12   return;
13 }
```

Poboljšanje algoritma *Insertion sort*

Jedan od **problema** kod običnog algoritma **Insertion sort** je što **pomaci** elemenata idu za po **jedno** mjesto (u navedenim algoritmima $\rightarrow$).

Poboljšanje: povećati udaljenost (**razmak** za **pomake**) tako da sortiramo **više** potpolja s **većom** udaljenošću elemenata.

Navedeno poboljšanje se realizira korištenjem **niza padajućih** razmaka. Prvo se sortiraju **daleki** elementi, zatim **nešto bliži** pa **susjedi**.

Dobiveni algoritam sortiranja se zove **Shell sort** i ime je dobio po autoru *Donald L. Shell* (1959. godine).
Originalni niz **razmaka** (broj **manjih** polja za sort): $n/2$ polja s ≈ 2 elementa, $n/4, \ldots, 2, 1$ (cijelo polje).

Analiza **složenosti** je vrlo **komplicirana**. Uz pravi izbor niza **razmaka**, algoritam može biti bitno **brži** od **kvadratnog**.

Donja ograda za složenost sortiranja usporedbom

Može se pokazati da za **proizvoljan** algoritam sortiranja koji koristi **usporedbe** parova članova (**binarne** relacije $<, >$) mora biti:

$$\text{broj\_usporedbi} \geq c \cdot n \cdot \log n$$

gdje je c neka pozitivna konstanta. Broj usporedbi je reda veličine **barem** $n \cdot \log n$. To je bitno brže od n^2 za veće brojeve n. **Dobra** stvar je što **postoje** algoritmi sortiranja s takvim redom veličine broja usporedbi.

Postoje i **brži** algoritmi sortiranja (ne koriste usporedbe) ali rade samo za **specijalne** vrste podataka. Npr. **Radix sort** radi za **nenegativne brojeve** i ima **linearnu** složenost (međutim nužno koristi dodatno polje - ovisno o implementaciji, dodatno polje može imati **manje ili jednako** n elemenata).

Algoritmi sortiranja u praksi

Algoritmi sortiranja koji se **koriste** u praksi:

- **QuickSort** - autor *C. A. R. (Tony) Hoare*, 1962. godine. **Prosječna** složenost mu je reda veličine $n \cdot \log n$ za slučajne (dobro razbacane) nizove. U najgorem slučaju, složenost je reda veličine n^2. Algoritam se koristi zbog dobre **prosječne** brzine i dio je **standardne** `C` biblioteke.
- **HeapSort** - autor *John W. J. Williams*, 1964. godine. **Prosječna** i **najgora** složenost su reda veličine $n \cdot \log n$. U **prosjeku** je nešto **sporiji** od **QuickSort**-a.
- **MergeSort** - sortiranje **sortiranim spajanjem**. **Najgora** složenost je $n \cdot \log n$. Lakše se realizira na **vezanoj listi** nego na **polju**. Ukoliko se primjenjuje na **polju**, potrebno je **pomoćno** polje i dodatna kopiranja između ta **dva** polja. Autor: *John von Neumann*, 1945. godine. **Prvi** program za računalo koje **sprema** i podatke i **programe** (von Neumannov model). Računalo se zvalo `EDVAC`.

Usporedba algoritama sortiranja kvadratne složenosti

Vrijeme (u s) za sortiranje polja s $n = 10^5$ elemenata:

Algoritam	Slučajno	Uzlazno	Silazno
`sel_sort`	1.763	1.766	1.765
`sel_sort_f2`	1.765	1.755	1.861
`sel_sort_max`	1.422	1.425	1.423
`sel_sort_f3`	1.424	1.425	1.428
`bubb_sort`	12.064	2.154	5.367
`bubb_sort_f1`	11.952	0.000	5.361
`bubb_sort_f2`	12.427	0.000	5.320
`ins_sort`	1.212	0.000	2.431
`ins_sort_st`	1.191	0.000	2.383
`ins_sort_bs`	0.822	0.000	1.627

Bubble sort u praktičnim primjenama ima **najgore** vrijeme izvršavanja.

Usporedba algoritama sortiranja prosječne složenosti reda $\leq n \cdot \log n$

Vrijeme (u s) za sortiranje polja s $n = 10^6$ elemenata:

Algoritam	Slučajno	Uzlazno	Silazno
`quick_sort`	0.076	1.442	1.441
`heap_sort`	0.102	0.042	0.048
`merge_sort`	0.098	0.021	0.022
`shell_sort`	0.116	0.009	0.016
`radix_sort`	0.015	0.026	0.025

Quick sort u praktičnim primjenama ima **najbolje** vrijeme izvršavanja od **generalno** primjenjivih algoritama.