Sortowanie

- Ustawienie rekordów w określonej kolejności, np. alfabetycznej lub numerycznej, od najmniejszego do największego

Tablica przed sortowaniem

Indeks	Wartość
[0]	15
[1]	128
[2]	3
[3]	27
[4]	32
[5]	56
[6]	1
[7]	87

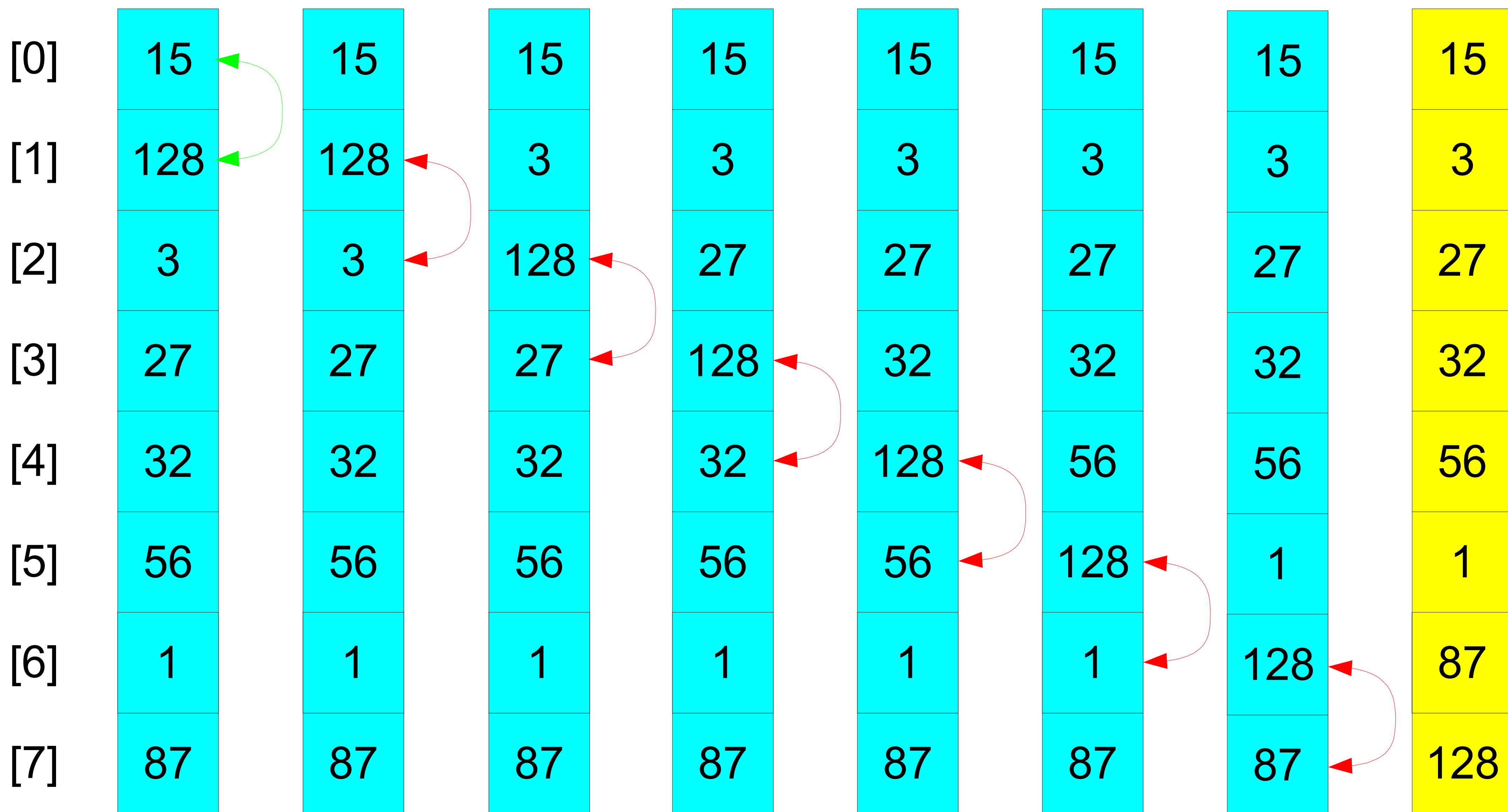
Tablica po sortowaniu

Indeks	Wartość
[0]	1
[1]	3
[2]	15
[3]	27
[4]	32
[5]	56
[6]	87
[7]	128

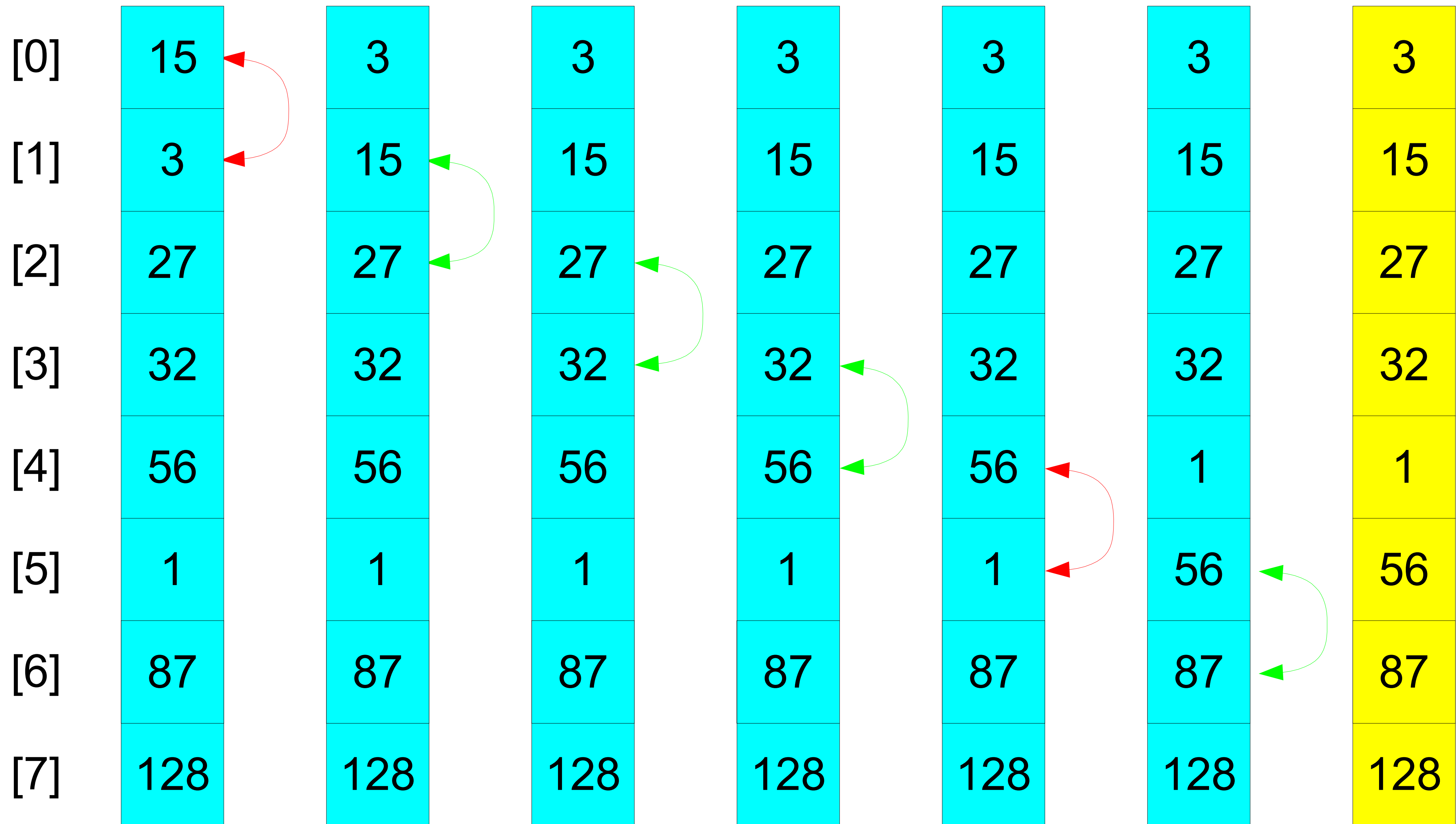
Sortowanie bąbelkowe

- Sortowanie bąbelkowe to jeden z najprostszych (i najmniej efektywnych) algorytmów sortowania.
- Przy sortowaniu bąbelkowym porównuje się dwie sąsiednie wartości i zamienia je w razie potrzeby miejscami w celu uzyskania właściwej kolejności. Jest wiele wersji algorytmu, ale w każdej porównuje się pary sąsiadujących elementów.
- Większe wartości stopniowo unoszą się w górę jak bąbelki powietrza w wodzie.
- W celu ukończenia sortowania konieczne trzeba wielokrotnie przejść przez całą tablicę.

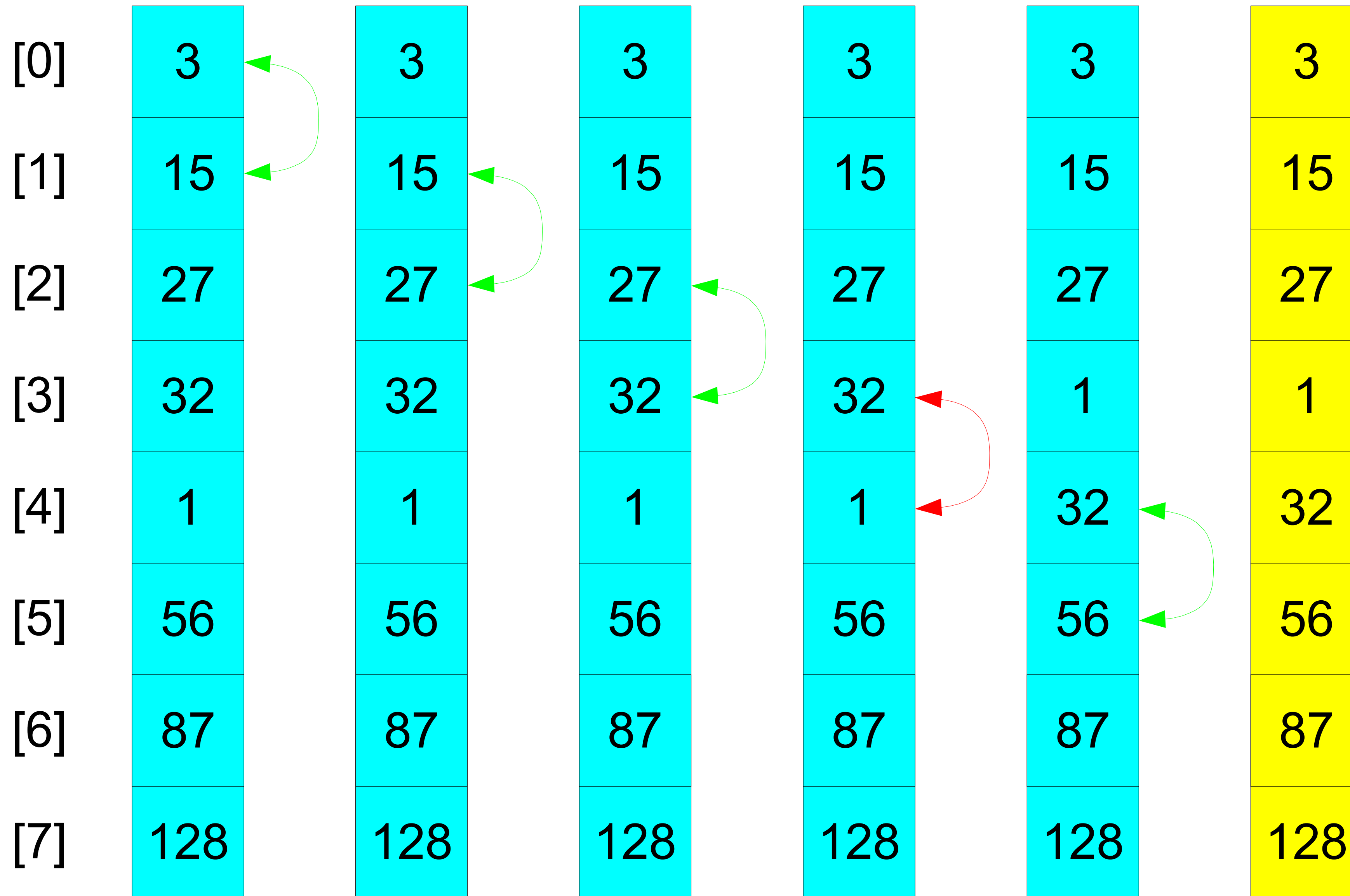
Bubblesort (przebieg 1)



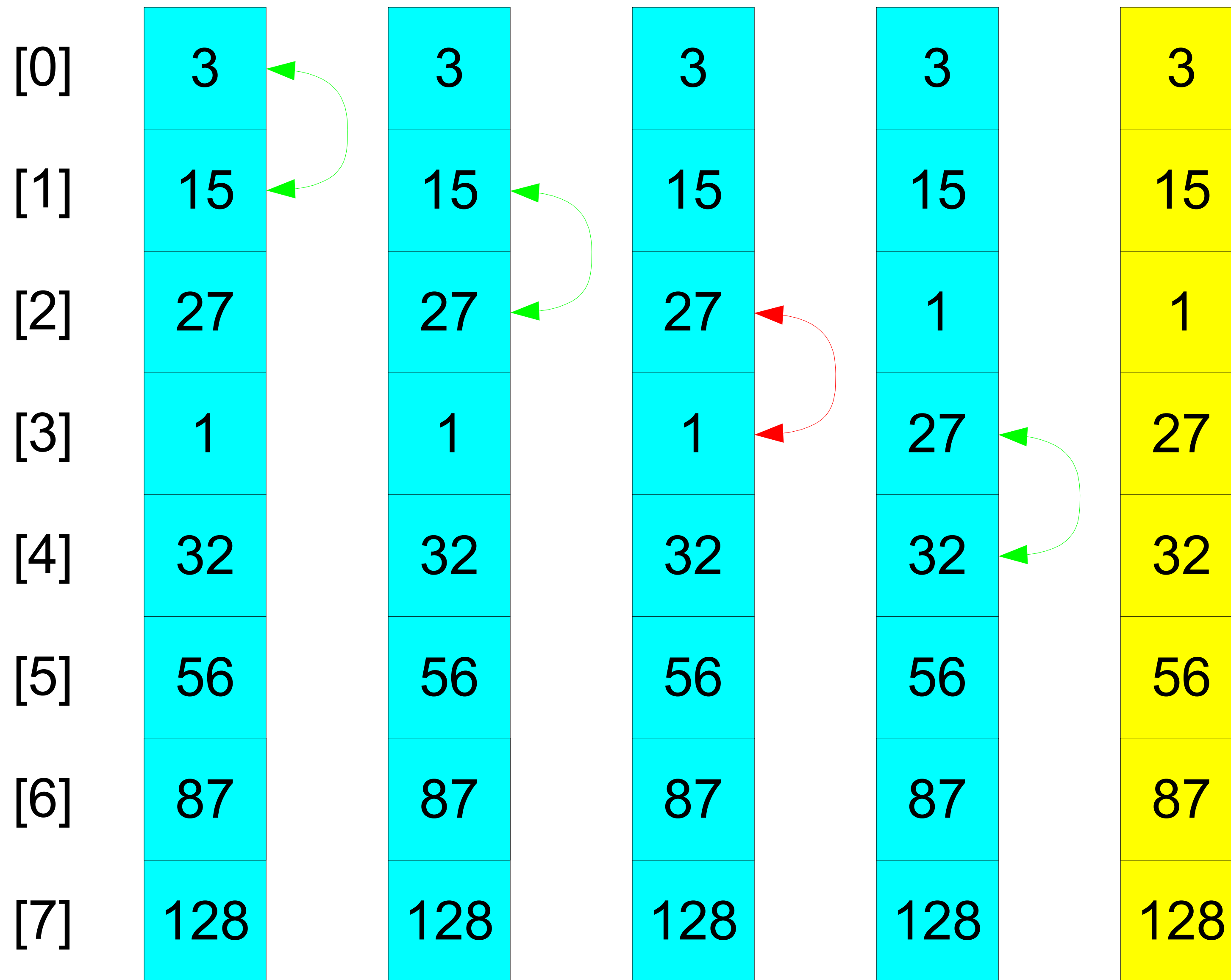
Bubblesort (przebieg 2)



Bubblesort (przebieg 3)

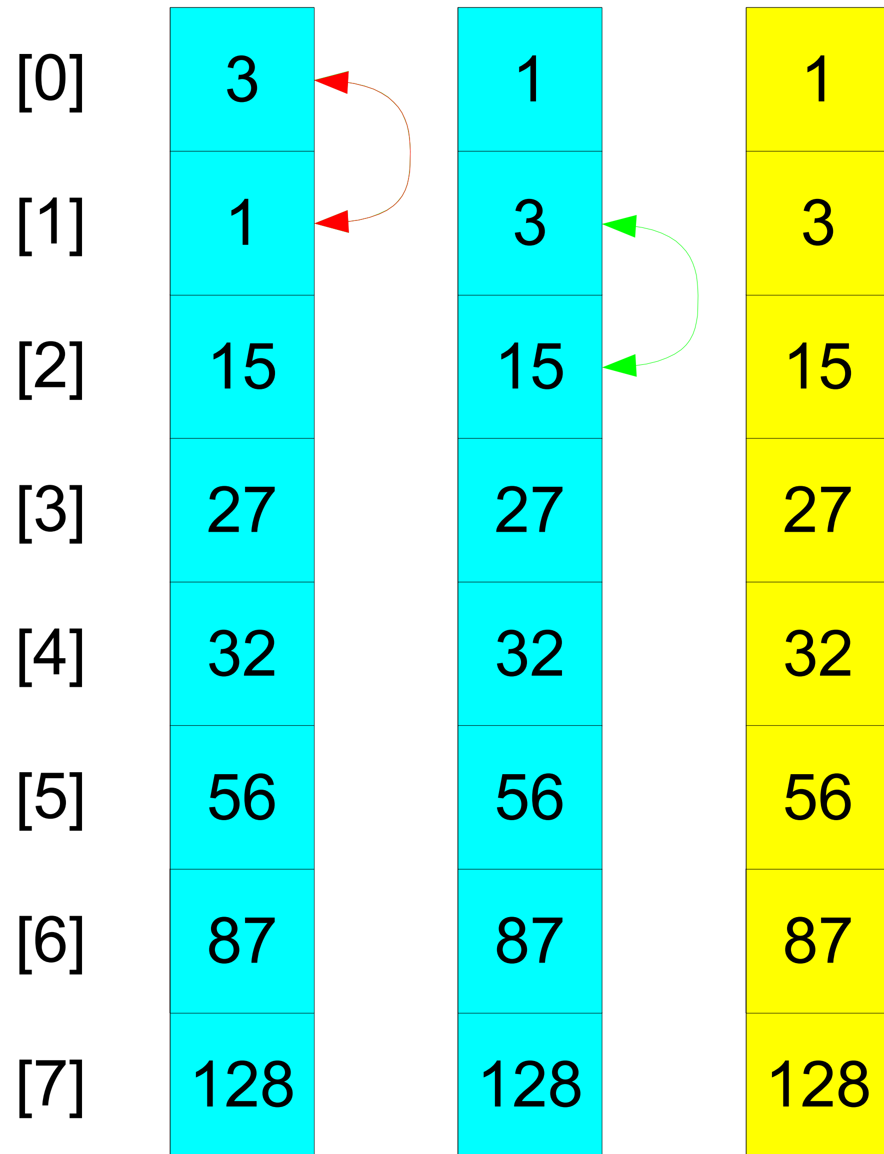


Bubblesort (przebieg 4)



Bubblesort (przebieg 5)

Bubblesort (przebieg 6)



Bubblesort (przebieg 7)

[0]	1	1
[1]	3	3
[2]	15	15
[3]	27	27
[4]	32	32
[5]	56	56
[6]	87	87
[7]	128	128

Bubblesort - program w C

```
#include <stdio.h>

void display (int *a, unsigned int n) {
  unsigned int i;
  printf ("Printing array\n");
  for (i = 0; i < n; i++)
    printf ("%d\n", a[i]);
}

void swap (int *a, int *b) {
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}

void bubblesort (int *a, unsigned int n) {
  unsigned int i, j;
  for (i = n - 1; i > 0; i--)
    for (j = 0; j < i; j++)
      if (a[j] > a[j + 1])
        swap (a + j, a + j + 1);
}

int main () {
  int numbers[] = { 15, 128, 3, 27, 32, 56, 1, 87 };
  display (numbers, sizeof (numbers) / sizeof (*numbers));
  bubblesort (numbers, sizeof (numbers) / sizeof (*numbers));
  display (numbers, sizeof (numbers) / sizeof (*numbers));
  return 0;
}
```

Bubblesort - program w C dla kont klientów

- W celu sortowania kont klientów należy zmienić funkcję zamieniającą i porównującą

```
typedef struct
{
  char name[21];
  float balance;
  int accNo;
}
CustAccount;

void
swap (CustAccount * a, CustAccount * b)
{
  CustAccount tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}

void
bubblesort (CustAccount * a, unsigned int n)
{
  unsigned int i, j;
  for (i = n - 1; i > 0; i--)
    for (j = 0; j < i; j++)
      if (strcmp (a[j].name, a[j + 1].name) > 0)
        swap (a + j, a + j + 1);
}
```

Bubblesort - program w C dla kont klientów

- Wskaźnik do funkcji porównującej może być parametrem funkcji sortującej w celu umożliwienia różnych kryteriów sortowania

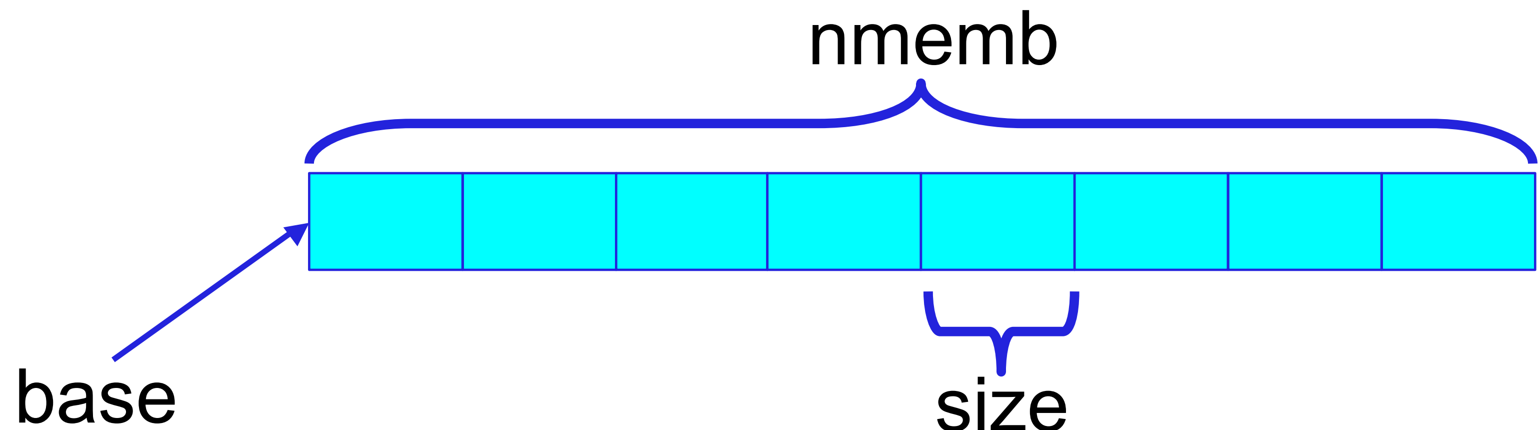
```
void swap (CustAccount * a, CustAccount * b) {
  CustAccount tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}

void bubblesort (CustAccount * a, unsigned int n,
            int (*comp) (const CustAccount *, const CustAccount *)) {
  unsigned int i, j;
  for (i = n - 1; i > 0; i--)
    for (j = 0; j < i; j++)
      if (comp (a + j, a + j + 1) > 0)
        swap (a + j, a + j + 1);
}

int compAccNo (const CustAccount * p1, const CustAccount * p2) {
  return p1->accNo - p2->accNo;
}

int compName (const CustAccount * p1, const CustAccount * p2) {
  return strcmp (p1->name, p2->name);
}
```

```
CustAccount accounts[] = { ... };
bubblesort (accounts, sizeof (accounts) / sizeof (*accounts), compName);
bubblesort (accounts, sizeof (accounts) / sizeof (*accounts), compAccNo);
```

Bubblesort - przypadek ogólny

- Aby przeprowadzić sortowanie w ogólnym przypadku, **`bubblesort`** potrzebuje dwóch informacji:
  - Rozmiaru pokedynczego elementu tablicy
  - Kryterium porównania
- Rozmiar rekordu jest również potrzebny funkcji **`swap`**

```
void bubblesort (void *base, unsigned int nmemb, unsigned int size,
                 int (*comp) (const void *, const void *)) {
  unsigned int i, j;
  for (i = nmemb - 1; i > 0; i--)
    for (j = 0; j < i; j++)
      if (comp ((char *) base + j * size, (char *) base + (j + 1) * size) > 0)
        swap ((char *) base + j * size, (char *) base + (j + 1) * size, size);
}
```



Bubblesort - swap

- **`swap`** zamienia miejscami zawartość dwóch obszarów pamięci, niezależnie od ich typu
- Prosta implementacja jest przedstawiona poniżej

```
void swap (void *a, void *b, unsigned int size) {
  char tmp;
  unsigned int i;
  for (i = 0; i < size; i++)
    {
      tmp = *((char *) a + i);
      *((char *) a + i) = *((char *) b + i);
      *((char *) b + i) = tmp;
    }
}
```

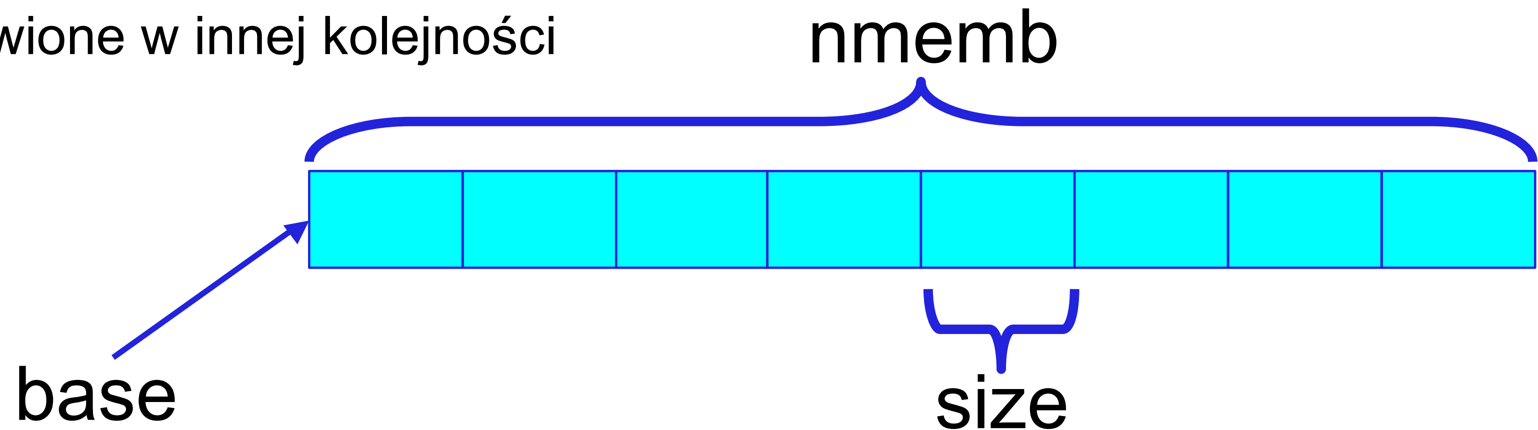
Bubblesort - przykład

```
int compAccNo (const void *p1, const void *p2) {
  return ((CustAccount *) p1)->accNo - ((CustAccount *) p2)->accNo;
}

int compName (const void *p1, const void *p2) {
  return strcmp (((CustAccount *) p1)->name, ((CustAccount *) p2)->name);
}

int compNum (const void *p1, const void *p2) {
  return *((int *) p1) - *((int *) p2);
}
```

```
CustAccount accounts[] = { ... };
int numbers[] = { 15, 128, 3, 27, 32, 56, 1, 87 };

bubblesort (accounts, sizeof (accounts) / sizeof (*accounts),
            sizeof (*accounts), compName);

bubblesort (accounts, sizeof (accounts) / sizeof (*accounts),
            sizeof (*accounts), compAccNo);

bubblesort (numbers, sizeof (numbers) / sizeof (*numbers),
            sizeof (*numbers), compNum);
```

`qsort`

- **`void qsort(void * base, size_t nmemb, size_t size, int (* compar)(const void *, const void *));`**
  - **`qsort`** sortuje tablicę **`nmemb`** obiektów zaczynającą się pod adresem **`base`**.
  - **`size`** określa rozmiar pojedynczego elementu tablicy.
  - Wskaźnik do funkcji porównującej przekazywany jest jako argument **`compar`**
  - Pozwala to na sortowanie obiektów o dowolnych właściwościach
    - **`compar`** powinna przyjmować dwa argumenty, każdy będący wskaźnikiem do elementu tablicy
    - Wynik **`compar`** musi być ujemny jeżeli pierwszy argument jest mniejszy od drugiego, zerowy jeżeli argumenty są równe i dodatni jeżeli pierwszy argument jest większy niż drugi (gdzie *mniejszy od* i *większy niż* odnoszą się do wybranego kryterium sortowania).
  - Tablica jest sortowana w miejscu, tj. po powrocie z funkcji **`qsort`** elementy tablicy **`base`** są ustawione w innej kolejności

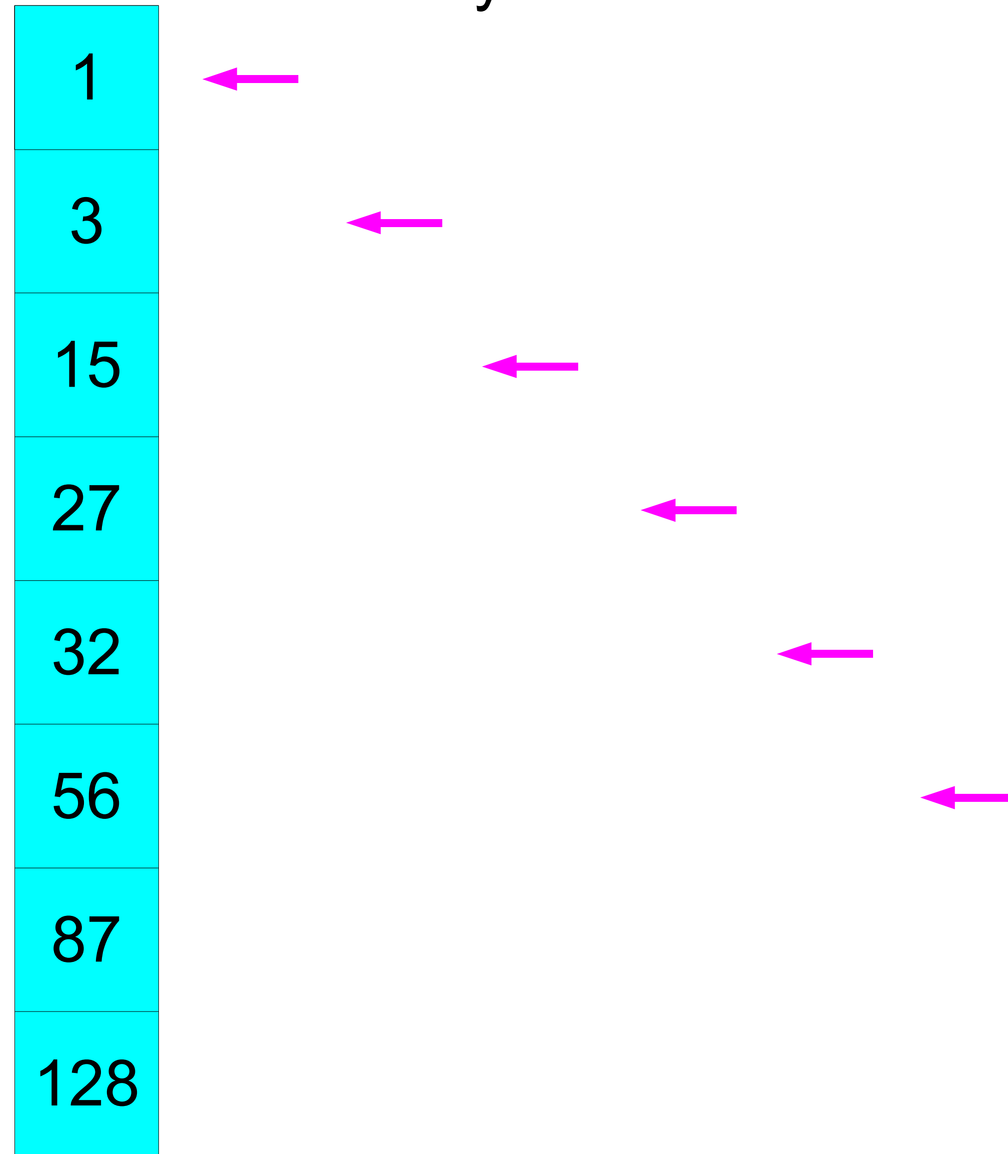


qsort - przykład

```
typedef struct {
  char name[21];
  float balance;
  int accNo;
} CustAccount;

CustAccount accounts[] = { ... };

int compAccNo (const void *p1, const void *p2) {
  return ((CustAccount *) p1)->accNo - ((CustAccount *) p2)->accNo;
}

int compName (const void *p1, const void *p2) {
  return strcmp (((CustAccount *) p1)->name, ((CustAccount *) p2)->name);
}

int main()
{
  ...

  /* sort by name */
  qsort (accounts, sizeof (accounts) / sizeof (*accounts), sizeof (*accounts),
         compName);


  /* sort by accNo */
  qsort (accounts,
         sizeof (accounts) /
         sizeof (*accounts), sizeof (*accounts), compAccNo);
  ...

}
```

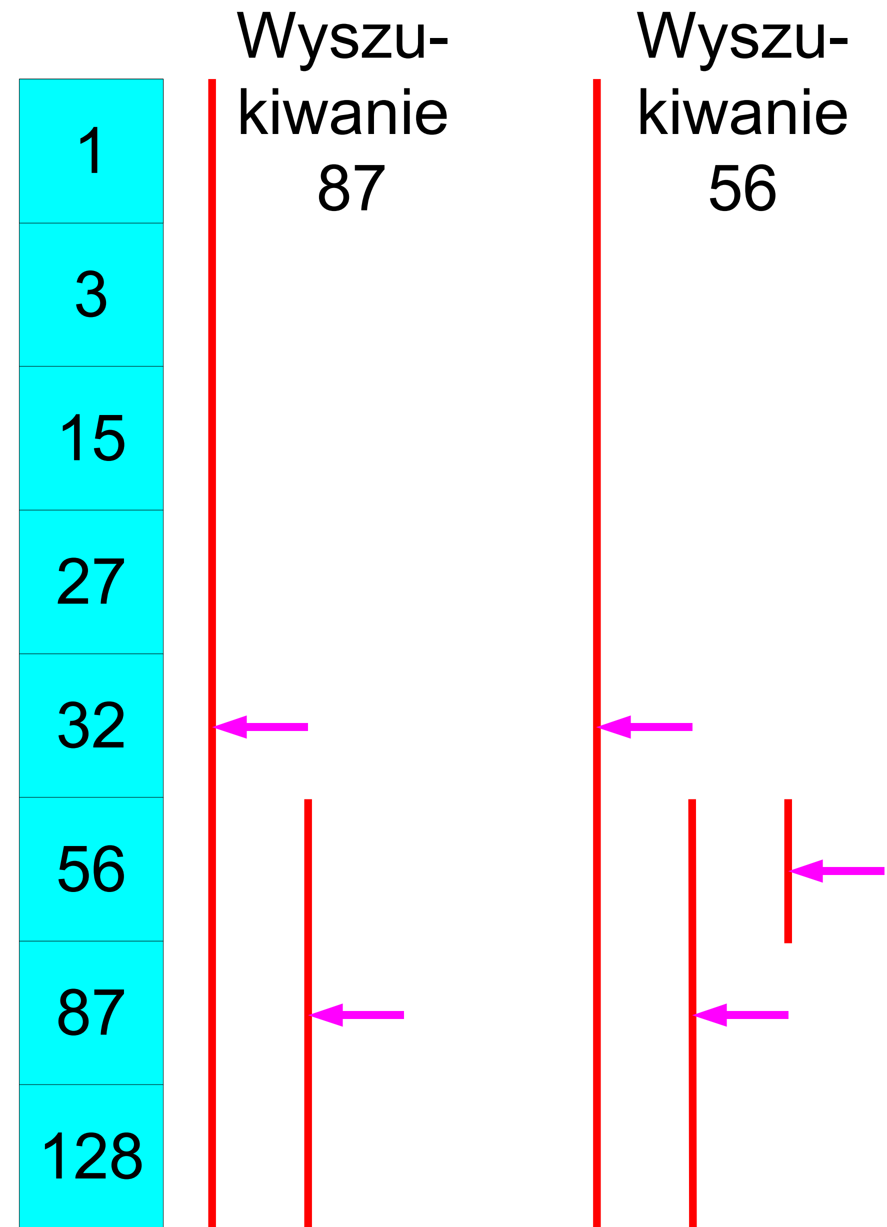
Wyszukiwanie liniowe

- Zaczynamy od początku tablicy i porównujemy po kolei wszystkie elementy



Wyszukiwanie binarne

- Pozwala na szybkie wyszukiwanie w posortowanej tablicy
- Zaczynamy od środka i określamy, w której połowie szukać dalej
- Powtarzamy dopóki nie znajdziemy klucza lub pozostały obszar będzie pusty



Wyszukiwanie binarne - program w C

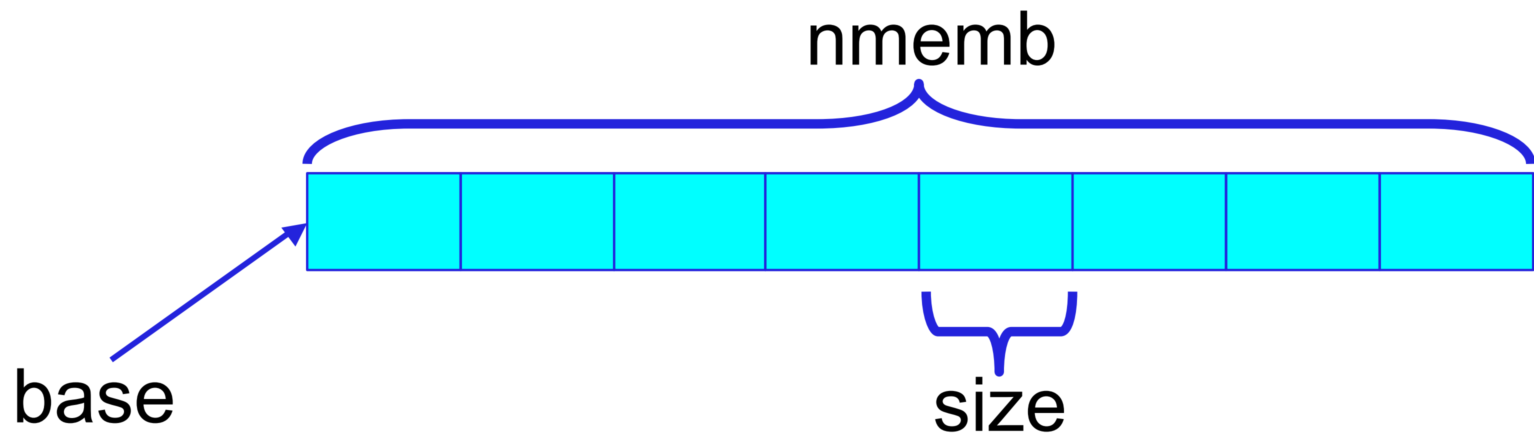
```
int * binarySearch (int what, int *low, int *high) {
  while (high > low)
    {
      int *middle = low + (high - low) / 2;
      if (*middle == what)
        return middle;
      else if (*middle > what)
        high = middle;
      else
        low = middle + 1;
    }
  return NULL;
}

void search (int number, int *low, int *high) {
  if (binarySearch (number, low, high))
    printf ("%d found\n",number);
  else
    printf ("%d not found\n",number);
}

int main () {
  int numbers[] = { 1, 3, 15, 27, 32, 56, 87, 128 };
  search (87, numbers, numbers + sizeof (numbers) / sizeof (*numbers));
  search (56, numbers, numbers + sizeof (numbers) / sizeof (*numbers));
  search (16, numbers, numbers + sizeof (numbers) / sizeof (*numbers));
  return 0;
}
```

Wskaźnik do komórki pamięci zaraz za ostatnim elementem

Wskaźnik do pierwszego elementu tablicy

bsearch

- `void *bsearch(const void *key, const void *base,`
  `size_t nmemb, size_t size,`
  `int (*compar)(const void *, const void *));`
  - **bsearch** przeszukuje tablicę **nmemb** obiektów, rozpoczynającą się od adresu **base**, w poszukiwaniu rekordu zgodnego z obiektem wskazywanym przez **key**. Rozmiar rekordu jest przekazywany jako parametr **size**.
  - Zawartość tablicy powinna być posortowana w kolejności wyznaczonej przez funkcję porównującą **compar**.
  - **bsearch** zwraca wskaźnik do znalezionego rekordu lub **NULL** gdy rekordu nie znaleziono. Jeżeli tablica zawiera wiele elementów zgodnych z wzorcem, nie jest określone który zostanie znaleziony.



bsearch - przykład

```
typedef struct {
  char name[21];
  float balance;
  int accNo;
} CustAccount;

CustAccount accounts[SIZE];

int compAccNo (const void *p1, const void *p2) {
  return ((CustAccount *) p1)->accNo - ((CustAccount *) p2)->accNo;
}

int compName (const void *p1, const void *p2) {
  return strcmp (((CustAccount *) p1)->name, ((CustAccount *) p2)->name);
}

CustAccount * findByName (const char *name) {
  CustAccount reference;
  strcpy (reference.name, name);
  return bsearch (&reference, accounts, sizeof (accounts) / sizeof (*accounts),
                  sizeof (*accounts), compName);
}

CustAccount * findByAccNo (int accNo) {
  CustAccount reference;
  reference.accNo = accNo;
  return bsearch (&reference, accounts, sizeof (accounts) / sizeof (*accounts),
                  sizeof (*accounts), compAccNo);
}
```