РЕШАВАНЕ НА ЗАДАЧИ ОТ АЛГОРИТМИ И СТРУКТУРИ ОТ ДАННИ В C

Мартин Велчев 11в клас,

23.11.2025 г.

РЕЗЮМЕ

Разглеждаме няколко задачи, свързани с алгоритми и структури от данни, и оптималните им решения спрямо материала, изучен досега в часовете по ОС.

Ключови думи: побитови операции, множества, структури от данни, битови маски, C програмиране, оптимизация на памет, алгоритми

Съдържание

1 МНОЖЕСТВО

Реализирайте структура от данни множество за естествени числа n $< 2^{13}$. Нека предоставеният интерфейс съдържа следните функции:

- *за добавяне премахване и проверка на съществуване на елемент (съответно insert, remove и contains);*
- *за обединение, сечение и разлика (съответно union, intersect и diff).*

Структура от данни

Задачата изисква да се направи структура от данни, която да съхранява естествени числа в интервала от 1 до 2^{13}, т. е. от 1 до 8192. За целта може да се използва сруктура, в която има масив от тип int с 2^{13} елемента:

```
struct set {
    int data[1 << 13];
};
```

Това обаче не би било ефективно, тъй като една променлива от тип int заема 4 байта в паметта, следователно ще заделим 4 байта * 8192 = 32768 байта. За да спестим място, можем да направим масива от тип char и да съхраняваме булева стойност, т.е. ако масива съдържа числото 24, то на 24-тия индекс ще съхраняваме единица.

```
struct set {
    char data[1 << 13];
};
```

С този метод заделяме 1 байт * 8192 = 8192 байта. Структурата заема значително по-малко място, но има начин да спестим и още. Тъй като стойността, която пазим е булева, то за нея е нужен само 1 бит място в паметта. Всеки char е 1 байт = 8 бита в паметта, тоест на една позиция в масива можем да държим 8 елемента. За целта нашият масив ще има 8 пъти по-малко позиции - 2^{13} / 8 = 2^{10} = 1024:

```
struct set {
    char data[1 << 10];
};
```

С тази финална конфигурация заделяме 1 байт * 1024 = 1024 байта. За сравнение, това е 32 пъти по-малко заета памет за една и съща функционалност отколкото във варианта с int масив.

Insert

След като имаме установена структура на множеството, можем да започнем имплементацията на методите. Първият от тях е insert, който приема множество и елемент който да добави:

```
void insertElement(struct set * s, unsigned x);
```

За да добавим елемента в масива, трябва да сложим 1 на съответстващия му бит в маската. За целта първо трябва да определим в кой байт (елемент) се намира. Това може да стане като го разделим целочислено на 8 (1 байт = 8 бита):

```
unsigned byte_offset = x / 8;
```

Вече имаме съответния байт. Следващата стъпка е да открием съответстващия бит, като разделим числото модулно на 8, получавайки отместването му в битове:

```
unsigned bit_offset = x % CHAR_BIT;
// CHAR_BIT - макро за броя битове в char
```

Един примерен байт в масива изглежда по този начин:

1	1	0	0	1	0	0	1

Всеки бит съответства на число. Ние вече сме открили битът, който съответства на нашето число. За да сложим единица в него можем да извършим побитова операция ИЛИ между байта и число, което съдържа единица на съответстващия байт (побитова маска):

1	1	0	0	1	0	0	1

|

0	0	0	0	0	1	0	0

=>

1	1	0	0	1	1	0	1

Маската, с която извършваме операцията е 1, изместено наляво със съответното битово изместване, което сме открили. Например, ако отместването ни е 3, то маската, с която ще извършим логическата операция е 1 « 3 = 100.

```
void insertElement(struct set * s, unsigned x) {
    unsigned byte_offset = x / 8, bit_offset = x % CHAR_BIT;
    s->data[byte_offset] |= 1 << bit_offset;
}
```

Пример

x = 51

byte offset = 51 / 8 = 6

=> Елементът ни се намира на 6-ия индекс в масива. Нека той изглежда така:

0	0	1	1	0	1	0	0

bit offset = 51 % 8 = 3

1 « 3 = 1000

0	0	1	1	0	1	0	0

|

0	0	0	0	1	0	0	0

=>

0	0	1	1	1	1	0	0

Remove

Премахването на число от множеството е малко по-сложна операция. Първата стъпка е същата като при добавянето - да намерим байтовото и битовото отместване на съответното число в масива:

```
unsigned byte_offset = x / 8, bit_offset = x % CHAR_BIT;
```

За да сложим 0 на съответния бит можем да използваме логическата операция импликация. А какво представлява тя? Означава се със знака -> и гласи, че при A -> B = Y, ако A = 1, то B задължително трябва да бъде едно за да е верен резултатът.

A	B	$A->B$
0	0	1
0	1	1
1	0	0
1	1	1

0 -> 0: A равно ли е на 1? Не => Резултатът е 1
0 -> 1: A равно ли е на 1? Не => Резултатът е 1
1 -> 0: A равно ли е на 1? Да => B равно ли е на 1? Не => Резултатът е 0
1 -> 1: A равно ли е на 1? Да => B равно ли е на 1? Да => Резултатът е 1
Логическата операция A -> B може да се сведе до (!A) | B:
(!0) | 0 = 1 | 0 = 1
(!0) | 1 = 1 | 1 = 1
(!1) | 0 = 0 | 0 = 0
(!1) | 1 = 0 | 1 = 1
Премахването на елемента става чрез следната операция:

```
s->data[byte_offset] = ~((~s->data[byte_offset]) | (1 << bit_offset));
```

~- Побитово инвертиране
| - Побитово ИЛИ
Операцията, която виждаме е побитово инверсирана импликация.
Защо и как работи тя? Нека разгледаме началното състояние на елемента от масива:

1	1	0	0	1	1	0	1

Битовото ни изместване е 3, следователно числото, което ще използваме е 100:

0	0	0	0	0	1	0	0

Нека извършим импликация между двете. Получаваме:

0	0	1	1	0	1	1	0

Забелязваме, че получаваме търсеният от нас байт, но инвертиран. Нека го инвертираме обратно:

1	1	0	0	1	0	0	1

Финалният вариант на функцията ни:

```
void removeElement(struct set * s, unsigned x) {
    unsigned byte_offset = x / 8, bit_offset = x % CHAR_BIT;
    s->data[byte_offset] = ~((~s->data[byte_offset]) | (1 << bit_offset));
}
```

Пример
x = 51
byte offset = 51 / 8 = 6
=> Елементът ни се намира на 6-ия индекс в масива. Нека той изглежда така:

0	0	1	1	1	1	0	0

bit offset = 51 % 8 = 3
1 « 3 = 1000

0	0	1	1	1	1	0	0

->

0	0	0	0	1	0	0	0

=>
~

1	1	0	0	1	0	1	1

=>

0	0	1	1	0	1	0	0

Contains

Третият метод е може би най-лесният за имплементация. Изисква се от нас да проверим дали даден елемент съществува в дадено множество. Можем по вече познатия ни начин да открием съответстващия му бит в маската:

```
unsigned byte_offset = x / 8, bit_offset = x % CHAR_BIT;
```

Следващата стъпка е да проверим дали побитово И с отместената му единица ще даде резултат 1:

```
return (s->data[byte_offset] & (1 << bit_offset)) != 0;
```

Финалният вариант на функцията ни:

```
char containsElement(struct set * s, unsigned x) {
    unsigned byte_offset = x / 8, bit_offset = x % CHAR_BIT;
    return (s->data[byte_offset] & (1 << bit_offset)) != 0;
}
```

Пример
x = 51
byte offset = 51 / 8 = 6

=> Елементът ни се намира на 6-ия индекс в масива. Нека той изглежда така:

0	0	1	1	1	1	0	0

bit offset = 51 % 8 = 3
1 « 3 = 1000

0	0	1	1	1	1	0	0

&

0	0	0	0	1	0	0	0

=>

0	0	0	0	1	0	0	0

Резултатът от операцията е 1000, което не е равно на 0, съответно елементът присъства в множеството.

Unite

Следващите три метода представляват операции между две множества. Първият от тях е обединение. В математиката обединение на две множества е множество, съдържащо всички елементи от двете множества. Отбелязва се със $\cup$:

$$A = \{1, 3, 5, 7, 8\}, \quad B = \{1, 2, 4, 6, 8\}, \quad A \cup B = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

Нашата функция ще приема като аргумент две множества и ще ги обединява в първото, т.е. ако функцията приеме A и B, то A = A $\cup$ B

Това може много лесно да се постигне, като се приложи побитово ИЛИ между всеки два съответстващи байта:

1	0	0	0	1	1	0	0

|

1	1	0	1	1	0	0	1

=>

1	1	0	1	1	1	0	1

По този начин резултатът има всеки елемент, присъствал в някое от двете множества. Имплементация на функцията:

```
void unite(struct set * p, struct set * q) {
    for(int i = 0; i < ELEMENT_COUNT; i++) {
        p->data[i] |= q->data[i];
    }
}
```

Intersect

Следващият метод е сечение. В математиката сечение на две множества е множество, съдържащо само елементите, които присъстват и в двете множества. Отбелязва се със $\cap$:

$$A = \{1, 3, 5, 7, 8\}, \quad B = \{1, 2, 4, 6, 8\}, \quad A \cap B = \{1, 8\}$$

Нашата функция ще приема като аргумент две множества и ще ги сече в първото, т.е. ако функцията приеме A и B, то A = A ∩ B

Това може много лесно да се постигне, като се приложи побитово И между всеки два съответстващи байта:

1	0	0	0	1	1	0	0

&

1	1	0	1	1	0	0	1

=>

1	0	0	0	1	0	0	0

По този начин резултатът има само елементите, присъствали и в двете множества. Имплементация на функцията:

```
void intersect(struct set * p, struct set * q) {
    for(int i = 0; i < ELEMENT_COUNT; i++) {
        p->data[i] &= q->data[i];
    }
}
```

Differ

Последният метод за задачата е разлика. В математиката разлика на две множества е множество, съдържащо само своите елементи, които не присъстват в другото множество. Отбелязва се със $\setminus$:

$$A = \{1, 3, 5, 7, 8\}, \quad B = \{1, 2, 4, 6, 8\}, \quad A \setminus B = \{3, 5, 7\}$$

Нашата функция ще приема като аргумент две множества и ще ги диференцира в първото, т.е. ако функцията приеме A и B, то A = A \ B

Това може много лесно да се постигне, като се за всеки байт се приложи побитово И между байта от първото и побитово инверсирания байт от второто множество:

1	0	0	0	1	1	0	0

&
~(

1	1	0	1	1	0	0	1

) =>

1	0	0	0	1	1	0	0

&

0	0	1	0	0	1	1	0

=>

0	0	0	0	0	0	1	0

По този начин резултатът има само елементите, присъствали в първото, но не и във второто множество. Имплементация на функцията:

```
for(int i = 0; i < ELEMENT_COUNT; i++) {
    p->data[i] &= ~q->data[i];
}
```

По-долу може да видите цялата получена програма:

```
#include <stdio.h>
#include <stdlib.h>

#define ELEMENT_COUNT 1 << 10

struct set {
    char data[ELEMENT_COUNT];
};

void insertElement(struct set * s, unsigned x) {
    unsigned byte_offset = x / 8, bit_offset = x % CHAR_BIT;
    s->data[byte_offset] |= 1 << bit_offset;
}

void removeElement(struct set * s, unsigned x) {
    unsigned byte_offset = x / 8, bit_offset = x % CHAR_BIT;
    s->data[byte_offset] = ~((~s->data[byte_offset]) | (1 << bit_offset));
}

char containsElement(struct set * s, unsigned x) {
    unsigned byte_offset = x / 8, bit_offset = x % CHAR_BIT;
    return (s->data[byte_offset] & (1 << bit_offset)) != 0;
}

void unite(struct set * p, struct set * q) {
    for(int i = 0; i < ELEMENT_COUNT; i++) {
        p->data[i] |= q->data[i];
    }
}

void intersect(struct set * p, struct set * q) {
    for(int i = 0; i < ELEMENT_COUNT; i++) {
        p->data[i] &= q->data[i];
    }
```

```
}

void differ(struct set * p, struct set * q) {
    for(int i = 0; i < ELEMENT_COUNT; i++) {
        p->data[i] &= ~q->data[i];
    }
}
```

2 СТЕК

Реализирайте структура от данни стек, която съхранява поредица от записи „един върху друг“. Интерфейсът трябва да предоставя следните операции:

- *добавяне на запис (push);*
- *премахване на запис (pop);*
- *преглед на горния елемент (peek);*
- *освобождаване на паметта, заделена за стека.*

Реализацията да се извърши в два варианта: стек с фиксиран размер и стек с динамичен размер.

Стек с фиксиран размер

При имплементация на стек с фиксиран размер използваме масив като носител на данните. Всеки елемент се добавя на върха на стека и се премахва от върха. За целта структурата съдържа брояч на текущия размер и масив за съхранение на елементите.

```
struct stack {
    size_t size;
    void * buffer[];
};
```

Този подход позволява бърз достъп до елементите по индекс, а операциите push и pop се извършват за време $O(1)$.

Основни функции:

```
struct stack * stack_init() {
    return calloc(1, sizeof(struct stack) + stack_capacity * sizeof(void *));
}

bool stack_push(struct stack * s, void * val) {
    if(s->size >= stack_capacity) return false;
    s->buffer[s->size++] = val;
    return true;
}

bool stack_pop(struct stack * s, void ** out) {
    if(!s->size) return false;
    *out = s->buffer[--s->size];
    return true;
}

bool stack_peek(struct stack * s, void ** out) {
    if(!s->size) return false;
    *out = s->buffer[s->size - 1];
```

```
    return true;
}

void stack_release(struct stack * s) {
    free(s);
}
```

Принцип на работа: - `push`: добавя елемента на текущата позиция `size` и увеличава брояча. - `pop`: намалява брояча `size` и връща елемента на върха. - `peek`: връща елемента на върха, без да го премахва.

Стек с динамичен размер

При динамичен стек използваме свързан списък като носител на данните. Всеки елемент съдържа указател към следващия и потребителските данни. Това позволява стекът да расте и да се свива динамично, без фиксиран капацитет.

```
struct stack_item {
    void * data;
    struct stack_item * next;
};

struct stack {
    size_t size;
    struct stack_item * top;
};
```

Основни функции:

```
struct stack * stack_init() {
    return calloc(1, sizeof(struct stack));
}

bool stack_push(struct stack * s, void * val) {
    struct stack_item * item = calloc(1, sizeof(*item));
    if(!item) return false;
    item->data = val;
    item->next = s->top;
    s->top = item;
    s->size++;
    return true;
}

bool stack_pop(struct stack * s, void ** out) {
    if(!s->top) return false;
    struct stack_item * popped = s->top;
    *out = popped->data;
    s->top = s->top->next;
    free(popped);
    s->size--;
    return true;
```

```
}

bool stack_peek(struct stack * s, void ** out) {
    if(!s->top) return false;
    *out = s->top->data;
    return true;
}

void stack_release(struct stack * s) {
    while(s->top) {
        struct stack_item * temp = s->top->next;
        free(s->top);
        s->top = temp;
    }
    free(s);
}
```

Принцип на работа: - Всеки нов елемент се добавя на върха на списъка (`push`). - При премахване (`pop`) връщаме данните на върха и изтриваме елемента. - `peek` връща горния елемент без да го премахва. - Паметта се освобождава, като обхождаме целия списък и изтриваме елементите.

Сравнение на двата подхода

- **Фиксиран стек:** бърз достъп, но ограничен капацитет. Паметта е заделена предварително.
- **Динамичен стек:** неограничен капацитет (ограничен само от RAM), но достъпът е малко по-бавен заради разпределената памет и указателите.

3 ОПАШКА

Реализирайте структура от данни опашка, която съхранява поредица от записи „един след друг“. Основните операции са:

- *добавяне на запис (`enqueue`);*
- *премахване на запис (`dequeue`);*
- *преглед на първия елемент (`front`);*
- *преглед на последния елемент (`back`);*
- *освобождаване на паметта.*

Реализацията да се извърши в два варианта: опашка с фиксиран размер и опашка с динамичен размер.

Опашка с фиксиран размер

При опашка с фиксиран размер използваме масив като носител на данните, подобно на фиксирания стек. Разликата е, че елементите се добавят в края и се премахват от началото на масива.

```
struct queue {
    size_t size;
    void * buffer[];
};
```

Обяснение на функциите:

`queue`$_i$*nit* Заделя непрекъснат блок памет за структурата и масива с фиксиран капацитет.

`queue`$_e$*nqueue* Добавя елемент в края на масива (`buffer[size]`), след което увеличава брояча `size`.

`queue`$_d$*equeue* Премахва първия елемент от масива (`buffer[0]`) и измества всички останали елементи с една позиция наляво чрез `memmove`. Това гарантира, че следващият елемент става новият „front“.

`queue`$_f$*ront* Връщат съответно първия и последния елемент в опашката, без да ги премахват.

`queue`$_r$*elease* Освобождава паметта за структурата и масива.

```
struct queue * queue_init() {
    return calloc(1, sizeof(struct queue) + sizeof(void *) * queue_capacity);
}

bool queue_enqueue(struct queue * s, void * val) {
    assert(s);
    s->buffer[s->size++] = val;
    return true;
```

```
}

bool queue_dequeue(struct queue * s, void ** out) {
    assert(s);
    if(!s->size) return false;
    *out = s->buffer[0];
    memmove(s->buffer, &s->buffer[1], --s->size * sizeof(void *));
    return true;
}

bool queue_front(struct queue * s, void ** out) {
    assert(s);
    if(!s->size) return false;
    *out = s->buffer[0];
    return true;
}

bool queue_back(struct queue * s, void ** out) {
    assert(s);
    if(!s->size) return false;
    *out = s->buffer[s->size - 1];
    return true;
}

void queue_release(struct queue * s) {
    free(s);
}
```

Особености: - Премахването (`dequeue`) е малко по-бавно от push/pop при стека, защото трябва да изместим всички останали елементи. - При малки капацитети и редки операции това е приемливо, но при големи опашки по-ефективен е динамичният вариант с указатели.

Опашка с динамичен размер

При динамичния вариант използваме свързан списък. Всеки елемент съдържа данни и указател към следващия. Структурата поддържа два указателя: `head` за първия елемент и `tail` за последния. Това позволява добавяне в края и премахване от началото за време $O(1)$.

```
struct queue_item {
    void * data;
    struct queue_item * next;
};

struct queue {
    struct queue_item * head, * tail;
};
```

Обяснение на функциите:

- **queue_init**: инициализира опашката (‘head‘ и ‘tail‘ = ‘NULL‘)
- **queue_enqueue**: добавя елемент в края; ако е празна, става и началото, и краят
- **queue_dequeue**: премахва началния елемент и връща данните; ако опашката стане празна, ‘tail‘ = ‘NULL‘
- **queue_front/back**: връща данните на началния или крайния елемент без да ги премахва
- **queue_release**: освобождава всички елементи и самата опашка

```
struct queue * queue_init() {
    return calloc(1, sizeof(struct queue));
}

bool queue_enqueue(struct queue * s, void * val) {
    assert(s);
    struct queue_item * item = calloc(1, sizeof(struct queue_item));
    if(!item) return false;
    item->data = val;
    if(s->tail) s->tail->next = item;
    else s->head = item;
    s->tail = item;
    return true;
}

bool queue_dequeue(struct queue * s, void ** out) {
    assert(s);
    if(!s->head) return false;
    struct queue_item * popped = s->head;
    *out = popped->data;
    s->head = s->head->next;
    free(popped);
    if(!s->head) s->tail = NULL;
    return true;
}

bool queue_front(struct queue * s, void ** out) {
    assert(s);
    if(!s->head) return false;
    *out = s->head->data;
    return true;
}

bool queue_back(struct queue * s, void ** out) {
    assert(s);
    if(!s->tail) return false;
    *out = s->tail->data;
    return true;
```

```
}

void queue_release(struct queue * s) {
    assert(s);
    while(s->head) {
        struct queue_item * temp = s->head;
        s->head = s->head->next;
        free(temp);
    }
    s->tail = NULL;
    free(s);
}
```

Сравнение на двата подхода

- **Фиксирана опашка:** лесна имплементация, но премахването изисква изместване на елементи. Капацитетът е ограничен.
- **Динамична опашка:** ефективно добавяне и премахване ($O(1)$), неограничен размер. Изисква повече памет за указатели.

Бележка: Динамичната опашка е предпочитана при големи или непредвидими количества данни, докато фиксираната е достатъчна при малки и известни размери.

4 СОРТИРАНЕ С QSORT

Цел Да сортираме масив от структури `student` според различни критерии: оценка по математика, оценка по математика + програмиране, години и име.

Функцията qsort

В стандартната библиотека на C е предоставена функцията `qsort`, която позволява сортиране на масиви от произволен тип. Нейният прототип е:

```
void qsort(void *base, size_t nitems, size_t size,
           int (*compar)(const void *, const void*));
```

Обяснение на параметрите:

- `base` — указател към началото на масива, който ще се сортира.
- `nitems` — броят на елементите в масива.
- `size` — размерът на един елемент от масива в байтове.
- `compar` — указател към функция, която сравнява два елемента. Функцията трябва да връща:
  - отрицателно число, ако първият елемент трябва да бъде преди втория в масива,
  - нула, ако са равни,
  - положително число, ако първият елемент трябва да дойде след втория в масива.

Пример на извикване:

```
qsort(students, studentCount, sizeof(struct student), comparator);
```

—

Comparator: сортиране спрямо оценка математика

```
int comparator(const void * a, const void * b) {
    struct student * studentA = (struct student *) a,
                   * studentB = (struct student *) b;

    return studentA->mathGrade < studentB->mathGrade ? -1 : 1;
}
```

Обяснение: Този компаратор сортира масива във възходящ ред по оценката по математика.

- Подадените аргументи се кастват към указатели към структури от тип ученик.
- Ако оценката на `studentA` е по-малка от `studentB`, функцията връща `-1`, което казва на `qsort`, че `studentA` трябва да предшества `studentB`.
- В противен случай връща `1`, т.е. `studentB` предшества `studentA`.

Comparator: сортиране по математика и програмиране

```
int comparator(const void * a, const void * b) {
    struct student * studentA = (struct student *) a,
                   * studentB = (struct student *) b;

    return studentA->mathGrade == studentB->mathGrade ?
        studentA->programmingGrade < studentB->programmingGrade ? -1 : 1
        :
        studentA->mathGrade < studentB->mathGrade ? -1 : 1;
}
```

Обяснение: Този компаратор въвежда и вторично сортиране:
- Подадените аргументи се кастват към указатели към структури от тип ученик.
- Основно сортиране по оценка по математика.
- Ако двете оценки по математика са равни, се използва оценката по програмиране.
- Тернарният оператор (`?:`) позволява кратко условно сравнение.
- Така се гарантира стабилно сортиране по два критерия.

Comparator: сортиране по години (низходящо)

```
int comparator(const void * a, const void * b) {
    struct student * studentA = (struct student *) a,
                   * studentB = (struct student *) b;

    return studentA->age > studentB->age ? -1 : 1;
}
```

Обяснение: Тук искаме да сортираме **низходящо** по години:
- Подадените аргументи се кастват към указатели към структури от тип ученик.
- Ако `studentA->age` е по-голям, той трябва да предшества `studentB`, затова връщаме `-1`.
- В противен случай връщаме `1`.
—

Comparator: сортиране по име (лексикографско, низходящо)

```
int comparator(const void * a, const void * b) {
    struct student * studentA = (struct student *) a,
                   * studentB = (struct student *) b;
    int result = 0;
    for(int i = 0; i < strlen(studentA->name) && i < strlen(studentB->name); i++) {
        char currA = studentA->name[i], currB = studentB->name[i];
        result = currA > currB ? -1 : currA == currB ? 0 : 1;
        if(result != 0) break;
    }

    return result == 0 ? strlen(studentA->name) > strlen(studentB->name) ? -1 : 1 :
}
```

Обяснение:

- Цикълът сравнява буквите на имената по ред, като използва **ASCII стойности** за лексикографска наредба.

- Ако буквите се различават, резултатът се връща веднага.

- Ако са еднакви до края на по-късото име, по-дългото име се поставя по-напред (за низходящ ред).

- Така имаме низходящо сортиране по име по лексикографска наредба.

Пример за използване на `qsort`

Следният пример показва как можем да сортираме масив от ученици чрез `qsort` и да отпечатаме резултата:

```
int main() {
    struct student students[] = {
        {
            .name = "Gosho",
            .age = 17,
            .mathGrade = 4.92,
            .programmingGrade = 5.48,
            .historyGrade = 6.00,
        },
        {
            .name = "Ivan",
            .age = 18,
            .mathGrade = 5.10,
            .programmingGrade = 4.85,
            .historyGrade = 5.40,
        },
        {
            .name = "Maria",
            .age = 17,
            .mathGrade = 4.92,
            .programmingGrade = 5.30,
            .historyGrade = 5.60,
        },
        {
            .name = "Petar",
            .age = 19,
            .mathGrade = 3.40,
            .programmingGrade = 4.20,
            .historyGrade = 4.80,
        },
        {
            .name = "Elena",
            .age = 18,
            .mathGrade = 5.95,
            .programmingGrade = 5.70,
            .historyGrade = 5.85,
        },
        {
```

```
        .name = "Nikolay",
        .age = 20,
        .mathGrade = 2.90,
        .programmingGrade = 3.75,
        .historyGrade = 4.10,
    },
    {
        .name = "Desislava",
        .age = 19,
        .mathGrade = 4.60,
        .programmingGrade = 5.20,
        .historyGrade = 5.00,
    },
    {
        .name = "Georgi",
        .age = 18,
        .mathGrade = 3.85,
        .programmingGrade = 4.95,
        .historyGrade = 4.50,
    },
    {
        .name = "Kristina",
        .age = 17,
        .mathGrade = 5.30,
        .programmingGrade = 5.65,
        .historyGrade = 5.90,
    },
    {
        .name = "Stoyan",
        .age = 20,
        .mathGrade = 4.10,
        .programmingGrade = 4.60,
        .historyGrade = 3.95,
    }
};

qsort(
    students,
    sizeof students / sizeof(struct student),
    sizeof(struct student),
    comparator
);

printStudents(
    students,
    sizeof students / sizeof(struct student)
);

return 0;
```

```
}
```

Обяснение:

- Дефинираме масив `students` с 10 елемента от тип `struct student`, като задаваме всички необходими атрибути (`name, age, mathGrade, programmingGrade, historyGrade`).
- `qsort` приема четири параметъра:
  - `students` — началото на масива.
  - `sizeof students / sizeof(struct student)` — броят на елементите в масива (10 в случая).
  - `sizeof(struct student)` — размерът на един елемент в байтове.
  - `comparator` — функцията за сравнение, която определя правилата за сортиране.
- След сортирането, функцията `printStudents` (която е дефинирана от нас) обхожда масива и отпечатва всеки ученик, като показва резултата от сортирането.
- По този начин можем лесно да сменяме критерия за сортиране, като просто подадем друг компаратор на `qsort`.

Обобщение

- `qsort` е универсален инструмент за сортиране на масиви от произволен тип. - Ключовият елемент е **функцията компаратор**, която дефинира критериите за сортиране. - Може да се използва за **еднокритериално** и **многокритериално** сортиране. - При низходящ ред или сложни критерии, компараторът трябва да връща правилно отрицателни и положителни стойности за `qsort`.

5 СОРТИРАНЕ С RADIX SORT

Целта на задачата е да реализираме алгоритъм за цифрово сортиране на цели числа без знак. Този алгоритъм има линейна сложност $O(n)$ и се основава на разглеждането на битовото представяне на числата. При него числата се групират в "кофи"според стойността на даден бит, започвайки от най-старшия бит.

Пример с 4-битови числа

Нека имаме числата: 5, 13, 2, 7, 3, 11. Двоичното им представяне е:

5	0101
13	1101
2	0010
7	0111
3	0011
11	1011

Стъпка 1: Старши бит (бит 3)

Разделяме числата според най-старшия бит: 0 -> лява "кофа"; 1 -> дясна "кофа".

Преди: | 0 | 1 | 0 | 1 |, | 1 | 1 | 0 | 1 |, ...

След: 0-битове вляво, 1-битове вдясно:

0	1	0	1
0	0	1	0
0	0	1	1
0	1	1	1
1	1	0	1
1	0	1	1

Стъпка 2: Следващ бит (бит 2)

Обхождаме отделно лявата и дясната кофа и пак групираме по стойност на бита.

Стъпка 3: Бит 1

Разделяме кофите по бит 1.

Стъпка 4: Бит 0

След последното разделяне всички числа са вече сортирани.

Анализ на сложността

- Брой битове: b
- Брой елементи: n

Алгоритъмът обхожда всеки елемент точно b пъти. Тъй като числата имат фиксиран размер, $b = \text{const}$. Следователно времевата сложност е $O(n)$.

Паметта за рекурсия е най-много b нива, което е константа. Следователно сложността по памет е $O(1)$ (ако се реализира на място).

Имплементация в C

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

#define BIT_IS_ZERO(x, pos) (!((x) & (1u << (pos))))
#define MSB_POS(type) ( (int) (sizeof(type) * CHAR_BIT - 1) )
#define IS_SIGN_BIT(pos) (pos) == MSB_POS(int)
#define IS_IN_LEFT_BUCKET(x, pos) IS_SIGN_BIT((pos)) ^ BIT_IS_ZERO((x), (pos))

bool radix_sort_helper(int * arr, const size_t size, const int pos) {
    if(size <= 1 || pos < 0) return true;

    char breakerPos = 0;
    for(int i = 0; i < size; i++) {
        if(IS_IN_LEFT_BUCKET(arr[i], pos)) {
            if(i != breakerPos) {
                int temp = arr[i];
                arr[i] = arr[breakerPos];
                arr[breakerPos] = temp;
            }
            breakerPos++;
        }
    }

    bool left = radix_sort_helper(arr, breakerPos, pos - 1);
    if(!left) return false;

    bool right = radix_sort_helper(arr + breakerPos, size - breakerPos, pos - 1);

    return left && right;
}

bool radix_sort(int * arr, size_t size) {
    return radix_sort_helper(arr, size, MSB_POS(int));
}

bool print_array(int * arr, size_t size) {
    printf("\n[ ");
    for(int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("]");
}

int main() {
    int arr[] = { 0, -3, 15, 6, 9, 48, -24, 19 };

    print_array(arr, 8);
```

```
    radix_sort(arr, 8);

    print_array(arr, 8);

    return 0;
}
```

Обяснение на кода

- `BIT_IS_ZERO(x, pos)`: Проверява дали даден бит `pos` на число `x` е 0.
- `MSB_POS(type)`: Намира позицията на най-старшия бит за даден тип (int).
- `IS_SIGN_BIT(pos)`: Проверява дали текущият бит е знакът на числото (за signed int).
- `IS_IN_LEFT_BUCKET(x, pos)`: Проверява дали елементът трябва да отиде в лявата „кофа“. За signed числа се използва XOR с бита на знака, за да се сортират отрицателните числа правилно.
- `radix_sort_helper`: Рекурсивна функция, която:
  1. Спира рекурсията, ако размерът на подмасива е 1 или битовата позиция < 0.
  2. Разделя масива на две части – тези, които попадат в лявата и дясната кофа според текущия бит. Елементите с 0 на този бит се поставят отляво, а тези с 1 – отдясно.
  3. Разменя елементи на място чрез временна променлива `temp`.
  4. Извиква се рекурсивно за лявата и дясната част за следващия бит.
- `radix_sort`: Връща сортиран масив, започвайки от най-старшия бит.
- `print_array`: Отпечатва текущото съдържание на масива.

Пример на побитово сортиране на един от етапите

Нека имаме числото 5 и 13 (4-битови):

$$5 = \boxed{0}\boxed{1}\boxed{0}\boxed{1}, \quad 13 = \boxed{1}\boxed{1}\boxed{0}\boxed{1}$$

След сортиране по най-старшия бит (бит 3):

$$\boxed{0}\boxed{1}\boxed{0}\boxed{1} \quad \text{(лява кофа)}, \quad \boxed{1}\boxed{1}\boxed{0}\boxed{1} \quad \text{(дясна кофа)}$$

След следващите битове, числата се подреждат постепенно, докато се получи окончателно:

$$2, 3, 5, 6, 9, 11, 13, 15, 19, 48$$

Алгоритъмът гарантира линейна сложност при фиксирана битова дължина на числата.

6 ЛЕСНИ ЗАДАЧКИ – РЕШЕНИЯ И АНАЛИЗ

6.1 Булева логика чрез NOR

Условие: Имплементирайте булевите операции `not`, `and`, `or`, `imply`, `equiv`, използвайки единствено функцията `nor`.

Код:

```
#include <stdio.h>

char nor(char x, char y) {
    return !(x || y);
}

char not(char x) {
    return nor(x, x);
}

char and(char x, char y) {
    return nor(not(x), not(y));
}

char or(char x, char y) {
    return not(nor(x, y));
}

char imply(char x, char y) {
    return or(not(x), and(x, y));
}

char equiv(char x, char y) {
    return and(imply(x, y), imply(y, x));
}
```

Обяснение:

- `nor` реализира операцията $\neg(x \vee y)$ чрез логическите оператори `||` и `!`.
- `not(x)` се реализира като `nor(x, x)`, защото $\neg(x \vee x) = \neg x$.
- `and(x, y)` следва дефиницята $x \wedge y = \neg(\neg x \vee \neg y)$.
- `or(x, y)` е просто отрицание на `nor`.
- `imply(x, y)` следва дефиницията $\neg x \vee y$ - ако x е 1, то задължително y трябва да е 1.
- `equiv(x, y)` представлява двупосочна импликация (ако x е 1, то y трябва да бъде 1, и ако y е 1, то x трябва да бъде 1) и проверява еквивалентност чрез две импликации.

—

6.2 Обръщане на низ на място

Код:

```
#include <stdio.h>
#include <string.h>

void strrot(char str[]) {
    int i = 0, j = strlen(str) - 1;
    while(i <= j) {
        char temp = str[i];
        str[i++] = str[j];
        str[j--] = temp;
    }
}
```

Обяснение:

- Използват се два индекса – начало и край.
- Символите се разменят чрез временна променлива.
- Низът се модифицира директно в паметта (in-place).
- Алгоритъмът има сложност $\mathcal{O}(n)$ и използва $\mathcal{O}(1)$ памет.

—

6.3 Втори най-голям елемент

Код:

```
#include <stdio.h>

#define MAX2(x, y) (x) > (y) ? (x) : (y)
#define MAX3(x, y, z) MAX2(MAX2((x), (y)), (z))

#define MIN2(x, y) (x) < (y) ? (x) : (y)
#define MIN4(x, y, z, t) MIN2(MIN2((x), (y)), MIN2((z), (t)))

#define SECONDMAX4(x, y, z, t) \
    MIN4(MAX3((x), (y), (z)), \
         MAX3((x), (z), (t)), \
         MAX3((x), (y), (t)), \
         MAX3((y), (z), (t)))

int second2max(int x, int y, int z, int t) {
    return SECONDMAX4(x, y, z, t);
}
```

Обяснение:

- Изчисляват се всички максимуми на тройки.
- Вторият най-голям е минимумът от тези максимуми.

- Макросите се разгъват на ниво препроцесор.
- Няма цикли или условни оператори в самата функция.

—

6.4 Проверка за валиден C идентификатор

Код:

```
#include <stdio.h>
#include <ctype.h>
#include <stdbool.h>

bool isValidIdent(const char * token) {
    enum { q0, q1, e } next_state = q0;

    for(; token && *token; ++token) {
        switch(next_state) {
            case q0:
                if(isalpha(*token) || *token == '_')
                    next_state = q1;
                else
                    next_state = e;
                break;

            case q1:
                if(isalnum(*token) || *token == '_')
                    next_state = q1;
                else
                    next_state = e;
                break;

            case e:
                return false;
        }
    }
    return next_state == q1;
}
```

Обяснение:

- Реализиран е краен автомат.
- `q0` – начало, `q1` – валиден идентификатор.
- Първият символ не може да е цифра.
- Използват се стандартните функции `isalpha`, `isalnum`.

—

6.5 Корен трети чрез бинарно търсене

Код:

```
#include <stdio.h>
#include <float.h>

#define POW3(x) (((x)*(x)*(x)) >= DBL_MAX ? DBL_MAX : ((x)*(x)*(x)))
#define ABS(x) ((x) < 0 ? -(x) : (x))
#define APPROX_EQUAL(a,b) (ABS((a)-(b)) <= DBL_EPSILON * ABS((a)+(b)))

double root3(double x) {
    double max = DBL_MAX;
    double min = 0;
    int negative = x < 0;
    double absX = ABS(x);

    while(1) {
        double curr = (max - min) / 2 + min;
        if(APPROX_EQUAL(POW3(curr), absX))
            return negative ? -curr : curr;
        if(POW3(curr) > absX) max = curr;
        else min = curr;
    }
}
```

Обяснение:

- Използва се бинарно търсене върху числов интервал.
- Работи се с абсолютна стойност.
- `DBL_EPSILON` дефинира точността.
- Сложност $\mathcal{O}(\log n)$.

—

6.6 Брой дни между две дати

Код:

```
#include <stdio.h>

#define ABS(x) ((x) < 0 ? -(x) : (x))

struct date {
    int year;
    int month;
    int day;
};

int days_serial(const struct date * d) {
    int y = d->year;
```

```
    int m = d->month;
    int day = d->day;

    if (m < 3) {
        y--;
        m += 12;
    }

    return 365*y + y/4 - y/100 + y/400
         + (153*(m - 3) + 2)/5
         + day - 1;
}

int days_between(const struct date * day1, const struct date * day2) {
    return ABS(days_serial(day2) - days_serial(day1));
}
```

Обяснение:

- Датата се преобразува в сериен номер.
- Формулата отчита високосни години.
- Разликата е абсолютната стойност на разликата на серийните дни.

—

6.7 Инвертиране на битове

Код:

```
#include <stdio.h>

unsigned invert(unsigned x, unsigned p, unsigned n) {
    return x ^ (((1 << n) - 1) << p);
}
```

Обяснение:

- Създава се маска от n единици - (1 « n) - 1 - ако n = 4, резултатът ще е 1111.
- Маската се измества на позиция p.
- XOR инвертира точно тези битове и запазва останалите - 1010 **1011** 1101 1001 ^ 0000 **1111** 0000 0000 = 1010 **0100** 1101 1001.

—

6.8 Закръгляне до кратно на дадено число

Код:

```
#include <stdio.h>

unsigned floor(unsigned x, unsigned p2) {
    return x - (x % p2);
```

```
}

unsigned ceil(unsigned x, unsigned p2) {
    return floor(x, p2) + p2;
}
```

Обяснение:

- `floor` премахва остатъка.
- `ceil` добавя едно кратно.
- Използва се чиста аритметика.

—

6.9 Преобразуване между число и низ

Код:

```
#include <stdio.h>
#include <string.h>

#define ABS(x) ((x) < 0 ? -(x) : (x))

void strprependchar(char * dest, char c) {
    memmove(dest + 1, dest, strlen(dest) + 1);
    dest[0] = c;
}

int itoa(int n, char s[]) {
    int absN = ABS(n);
    while(absN > 0) {
        char curr = absN % 10 + '0';
        strprependchar(s, curr);
        absN /= 10;
    }
    if(n < 0) strprependchar(s, '-');
    return 0;
}

int atoi(int * n, char s[]) {
    char negative = 0;
    *n = 0;

    for(int i = 0; s[i]; i++) {
        if(i == 0 && s[i] == '-') negative = 1;
        else if(s[i] < '0' || s[i] > '9') return 1;
        else *n = *n * 10 + (s[i] - '0');
    }
    if(negative) *n *= -1;
    return 0;
}
```

—

6.10 Балансирани скоби

Код:

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

bool balanced(const char * str) {
    char stack[1024];
    int top = 0;

    for (int i = 0; str[i]; i++) {
        switch (str[i]) {
            case '(':
            case '[':
            case '{':
                stack[top++] = str[i];
                break;

            case ')':
                if (top == 0 || stack[--top] != '(') return false;
                break;

            case ']':
                if (top == 0 || stack[--top] != '[') return false;
                break;

            case '}':
                if (top == 0 || stack[--top] != '{') return false;
                break;

            default:
                return false;
        }
    }
    return top == 0;
}
```

Обяснение:

- Използва се стек, реализиран чрез масив.
- Отварящите скоби се push-ват.
- Затварящите проверяват върха.
- Линеен алгоритъм $\mathcal{O}(n)$.

7 ДОМАШНА РАБОТА: КОНВЕРТИРАНЕ МЕЖДУ БРОЙНИ СИСТЕМИ (BASE2BASE)

Целта на задачата е да се реализира програма `base2base`, която конвертира число от една позиционна бройна система в друга. Програмата приема:

- начална бройна система (цяло число между 2 и 64),
- крайна бройна система (цяло число между 2 и 64),
- низ, представящ число в началната бройна система.

Поддържаните цифри за бройни системи до база 64 са:

- цифри: `0-9`
- малки латински букви: `a-z`
- главни латински букви: `A-Z`
- символи: `+` и `/`

Пълен код на програмата

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static const char DIGITS[] =
    "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ+/";

int getValue(char c) {
    const char *p = strchr(DIGITS, c);
    return (p ? (int)(p - DIGITS) : -1);
}

char getDigit(int value) {
    return DIGITS[value];
}

char *base2base(int sourceBase, int resultBase, const char *number) {
    if (sourceBase < 2 || sourceBase > 64 ||
        resultBase < 2 || resultBase > 64) {
        printf("Invalid base\n");
        return NULL;
    }

    unsigned long long base10 = 0;
    for (int i = 0; number[i]; i++) {
        int value = getValue(number[i]);
        if (value < 0 || value >= sourceBase) {
            printf("Invalid digit\n");
            return NULL;
        }
```

```
        base10 = base10 * sourceBase + value;
    }

    if (base10 == 0) {
        char *out = malloc(2);
        out[0] = '0';
        out[1] = '\0';
        return out;
    }

    char buffer[128];
    int pos = 0;

    while (base10 > 0) {
        int remainder = base10 % resultBase;
        buffer[pos++] = getDigit(remainder);
        base10 /= resultBase;
    }

    char *result = (char *) malloc(pos + 1);
    for (int i = 0; i < pos; i++) {
        result[i] = buffer[pos - 1 - i];
    }
    result[pos] = '\0';

    return result;
}

int main(int argc, char *argv[]) {
    char *r = base2base(atoi(argv[1]), atoi(argv[2]), argv[3]);
    if (r) {
        printf("%s\n", r);
        free(r);
    }
}
```

Обяснение на кода

Масивът DIGITS

```
static const char DIGITS[] =
    "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ+/";
```

Този масив дефинира всички позволени символи за бройни системи от база 2 до база 64. Индексът на даден символ в масива съответства на неговата числова стойност. Например:

- `'0'` $\rightarrow 0$
- `'a'` $\rightarrow 10$
- `'A'` $\rightarrow 36$
- `'/'` $\rightarrow 63$

Функция getValue

```
int getValue(char c) {
    const char *p = strchr(DIGITS, c);
    return (p ? (int)(p - DIGITS) : -1);
}
```

Функцията намира числовата стойност на даден символ:

- `strchr` търси символа `c` в низа `DIGITS`
- ако бъде намерен, връща указател към позицията му
- разликата `p - DIGITS` дава индекса, т.е. числовата стойност
- ако символът не съществува, връща `-1`

Функция getDigit

```
char getDigit(int value) {
    return DIGITS[value];
}
```

Тази функция е обратната на `getValue` – по дадена числова стойност връща символа, който я представя в съответната бройна система.

Проверка на бройните системи

```
if (sourceBase < 2 || sourceBase > 64 ||
    resultBase < 2 || resultBase > 64) {
    printf("Invalid base\n");
    return NULL;
}
```

Тук се гарантира, че и двете бройни системи са в позволения диапазон. При невалидна база функцията прекратява работа.

Преобразуване към база 10

```
unsigned long long base10 = 0;
for (int i = 0; number[i]; i++) {
    int value = getValue(number[i]);
    if (value < 0 || value >= sourceBase) {
        printf("Invalid digit\n");
        return NULL;
    }
    base10 = base10 * sourceBase + value;
}
```

Това е алгоритъм за превод от произволна бройна система към десетична:

- обхожда се всеки символ от входния низ
- извлича се неговата стойност
- текущото число се умножава по основата
- добавя се новата цифра

Специален случай: числото 0

```
if (base10 == 0) {
    char *out = malloc(2);
    out[0] = '0';
    out[1] = '\0';
    return out;
}
```

Ако числото е 0, се заделя динамична памет за низа `"0"` и той се връща директно.

Преобразуване от база 10 към целевата база

```
char buffer[128];
int pos = 0;

while (base10 > 0) {
    int remainder = base10 % resultBase;
    buffer[pos++] = getDigit(remainder);
    base10 /= resultBase;
}
```

Това е алгоритъм за превод от десетична към друга бройна система:

- взима се остатъкът при деление
- остатъкът определя последната цифра
- числото се дели на основата

Цифрите се получават в обратен ред и се записват временно в буфер.

Обръщане на реда и заделяне на резултата

```
char *result = (char *) malloc(pos + 1);
for (int i = 0; i < pos; i++) {
    result[i] = buffer[pos - 1 - i];
}
result[pos] = '\0';
```

Тук:

- се заделя точно толкова памет, колкото е нужна
- цифрите се копират в правилния ред
- добавя се терминираща нула

Функция main

```
int main(int argc, char *argv[]) {
    char *r = base2base(atoi(argv[1]), atoi(argv[2]), argv[3]);
    if (r) {
        printf("%s\n", r);
        free(r);
    }
}
```

- `atoi` конвертира аргументите от низ към цяло число
- извиква се функцията `base2base`
- резултатът се отпечатва
- заделената памет се освобождава с `free`

Заключение

Алгоритъмът работи на два етапа:

1. преобразуване към междинна база 10
2. преобразуване от база 10 към целевата база

Това позволява универсално конвертиране между произволни бройни системи в диапазона от 2 до 64.

ЛИТЕРАТУРА

[1] Remzi H. Arpaci-Dusseau и Andrea C. Arpaci-Dusseau. *Introduction to Operating Systems*. `https://pages.cs.wisc.edu/~remzi/OSTEP/intro.pdf`. Chapter from Operating Systems: Three Easy Pieces, Version 1.10. 2018.

[2] Remzi H. Arpaci-Dusseau и Andrea C. Arpaci-Dusseau. *Introduction to Operating Systems*. `https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-mechanisms.pdf`. Chapter from Operating Systems: Three Easy Pieces, Version 1.10. 2018.

[3] David Drysdale. *Anatomy of a system call, part 1*. `https://lwn.net/Articles/604287/`. Article from LWN.net. 2014.