Алгоритми и структуре података

2020/2021

Расподела поена

- Предиспитне обавезе
  - домаћи задаци (око 50 решених програмерских задатака) - 20 поена
  - тест - 10 поена
- Испит
  - практични испит
    - 3 задатка по 10 поена - праг 15 поена
  - усмени испит
    - 2 питања по 15 поена (са списка испитних питања) - праг 20 поена
    - додатни задатак (пример ван списка, који се решава техникама са списка)

Литература

- Филип Марић, Весна Маринковић, Младен Николић, Сана Стојановић-Ђурђевић: Алгоритми и структуре података, белешке са предавања и вежби
- Миодраг Живковић: Алгоритми
- Миодраг Живковић, Весна Маринковић: Алгоритми и структуре података, скрипта
- Предраг Јаничић, Филип Марић: Програмирање 2, скрипта
- Thomas H. Cormen. Charles E. Leiserson. Ronald L. Rivest. Clifford Stein. Introduction to Algorithms. Third Edition. The MIT Press
- Steven S. Skiena. The Algorithm Design Manual, Springer
- onlajn literatura (petlja.org, geeksforgeeks.org, . . .)

Предуслови

- Нема формалних предуслова, али се подразумева да сте савладали у потпуности:
  - градиво које се предаје на П1/П2,
  - елементарне појмове дискретне математике (који се покривају и у средњим школама).
- На усменом делу испита студент мора да влада свим овим појмовима (у супротном неће моћи да положи испит).
- У курсу ће се користити подскуп језика C++ (очекује се познавање језика C, а сви потребни појмови везани за језик C++ биће уведени на вежбама).

Алгоритми и структуре података

- Никлаус Вирт (творац програмског језика Паскал):

алгоритми + структуре података = програми

Алгоритми

- **Алгоритми** су средства за решавање прецизно дефинисаних рачунских проблема.
  - *Како се може прецизно дефинисати проблем сортирања?*
- *Примери*: алгоритам одређивања максимума серије елемената, алгоритам сортирања низа, алгоритам израчунавања вредности броја на основу његових цифара,...
- *Веома неформална дефиниција*: алгоритам је прецизно дефинисана процедура израчунавања тј. низ корака израчунавања, која, полазећи од неке вредности (или низа вредности, скупа вредности, итд.) као улаза производи неку вредност (или низ вредности, скуп вредности итд.) као излаз.

Формалне дефиниције појма алгоритма

Постоје прецизне дефиниције појма алгоритма (настале 1930-их када су се логичари бавили питањем шта се може, а шта не може израчунати праћењем прецизно дефинисаних поступака).

- URM машине
- Тјурингове машине
- Геделове рекурзивне функције
- Черчов ламбда рачун
- . . .

Опис алгоритма

Неки од најчешће коришћених механизама за опис алгоритама су следећи:

- програмски кôд
- псеудо-кôд
- блок дијаграми тока
- Scratch/Blockly дијаграми

Структуре података

- **Структуре података** представљају начине организације и складиштења података који омогућавају ефикасан приступ жељеним подацима и њихову ефикасну модификацију.
- Примери: низ, листа, балансирано бинарно дрво, стек, ред, ...
- Структуру података одређује:
  - колекција вредности података који се могу регистровати,
  - начин складиштења података у меморији и
  - функције и операције које се могу применити над тим подацима.

Два основна захтева

- Алгоритми (и пратећи програми) морају бити **коректни.**
- Алгоритми (и пратећи програми) морају бити довољно **ефикасни** да би могли да се примене на решавање реалних проблема.

Коректност алгоритама

Коректност алгоритама

- Иако се некада у пракси користе програми за које се зна да понекад могу да дају и нетачне резултате, то најчешће није случај и од програма се захтева да буде практично апсолутно непогрешив.

Спецификација

- Шта уопште значи да је алгоритам тј. програм коректан/исправан?
- Исправност програма почива на појму **спецификације**. Спецификација је, неформално, опис жељеног понашања програма који треба написати.
- Спецификација се обично задаје у терминима **предуслова** тј. услова које улазни параметри програма задовољавају, као и **постуслова** тј. услова које резултати израчунавања морају да задовоље.
    - Шта је спецификација проблема одређивања максимума низа бројева?

Парцијална и тотална коректност

- **парцијална коректност**: свака вредност коју алгоритам израчуна за улазне параметре који задовољавају спецификацију (тј. предуслов) мора да задовољи спецификацију (тј. постуслов).
- **заустављање**: алгоритам мора да се заустави за све улазе који задовољавају спецификацију (тј. предуслов).
- За заустављајуће парцијално коректне алгоритме кажемо да су **тотално коректни**.

Верификација

- Поступак показивања да је програм исправан назива се **верификовање програма**.
  - **динамичка верификација** подразумева проверу исправности у фази извршавања програма, најчешће путем тестирања;
  - **статичка верификација** подразумева анализу изворног кода програма, често коришћењем формалних метода и математичког апарата.

Извођење алгоритама из спецификације

- Синтеза програма
- Уместо да се накнадно верификује написани програм, програм се аутоматски изводи из дате спецификације (предуслова, постуслова и инваријанти)

Неке честе грешке у програмима

- Прекорачење распона типа бројева
- Прекорачење граница низа
- Лоше решени специјални случајеви
- ...

Индуктивно-рекурзивна конструкција

КЉУЧНА ИДЕЈА

Конструкција алгоритама веома тесно је повезана са доказивањем теорема математичком индукцијом

Математичка индукција

Фундаментална особина природних бројева

$$(P(0) \land (\forall n)(P(n) \Rightarrow P(n+1))) \Rightarrow (\forall n)(P(n))$$

- База индукције: $P(0)$
- Индуктивни корак: $(\forall n)(P(n) \Rightarrow P(n+1))$

Индуктивно-рекурзивни приступ

- Решење проблема веће димензије се проналази тако што решимо проблем/проблеме истог облика, али мање димензије и од решења тог/тих проблема добијемо решење полазног проблема веће димензије.
- За почетне димензије проблема решење морамо да израчунавамо директно, без даљег свођења на проблеме мање димензије.
- Ако се приликом свођења димензија проблема увек смањује, конструисани алгоритми ће се увек заустављати.

Имплементација претходне идеје

- **индуктивна (итеративна)**: променљивама унутар петље се итеративно ажурира вредност, кренувши од вредности које представљају решења елементарних проблема, па до крајњих вредности које представљају решења задатог проблема.
- **рекурзивна**: функција која решава полазни проблем сама себе позива да би решила проблем истог облика, али мање димензије (осим у случају елементарних проблема, који се директно решавају).

Пример - збир елемената низа

Итеративно

```
int zbir(int a[], int n) {
        int zbir = 0;
        for (int i = 0; i < n; i++)
                zbir = zbir + a[i];
        return zbir;
}
```

Рекурзивно

```
int zbir(int[] a, int n) {
        if (n == 0)
                return 0;
        else
                return zbir(a, n-1) + a[n-1];
}
```

Доказ коректности рекурзивних функција

- Индуктивна хипотеза је то да рекурзивни позив/позиви враћа/враћају коректне резултате и на основу тога се доказује да ће главни позив вратити коректан резултат.
- Базни случај је онај у коме функција не врши рекурзивни позив/позиве.

Пример

- Доказати коректност рекурзивне функције за одређивање минимума непразног низа бројева

```
int minNiza(int a[], int n) {
        if (n == 1)
                return a[0];
        else {
                int m = minNiza(a, n-1);
                return min(m, a[n-1]);
        }
}
```

Доказ коректности итеративних алгоритама - инваријанте петљи

- **Инваријанта петље** - логички услови који важе пре петље и након сваког извршавања наредби у телу петље, а након извршавања целе петље гарантују коректност алгоритма који та петља имплементира.
- Инваријанте суштински описују значење свих променљивих унутар петље.

Инваријанта петље

- Разматраћемо петље while, без наредби break и continue.

```
<incijalizacija>
// ovde vazi <invarijanta>
while (<uslov>)
        // ovde vaze i <uslov> i <invarijanta>
        <telo>
        // ovde vazi <invarijanta>
// ovde ne vazi <uslov>, a vazi <invarijanta>
```

Доказивање инваријантности услова

- Да бисмо доказали да је неки услов инваријанта петље, довољно је да докажемо:
  - (1) да тај услов важи пре првог уласка у петљу и
  - (2) да из претпоставке да тај услов важи пре неког извршавања тела петље и да је услов петље испуњен докажемо да тај услов важи и након извршавања тела петље.
- Та два тврђења нам гарантују да ће услов важити и након извршавања целе петље (ако се она икада заустави).
- Доказ је индукцијом по броју извршавања тела петље:
  - (1) омогућава доказ базног случаја
  - (2) омогућава доказ индуктивног корака

Доказивање коректности из инваријанте

- Свака петља има пуно инваријанти, међутим, од интереса су нам само оне инваријанте које у комбинацији са условом прекида петље имплицирају услов који нам је потребан након петље. Треба, дакле, доказати и наредни услов:
  - (3) из тога да инваријанта важи након завршетка петље и да услов петље није испуњен следи коректност алгоритма.

Пример

- Доказати коректност итеративне функције за одређивање минимума непразног низа бројева

```
int minNiza(int[] a, int n) {
        int m = a[0];
        for (int i = 1; i < n; i++)
                m = min(m, a[i]);
        return m;
}
```

Примена формалног метода

- Ретко када се у практичном програмирању коректност заиста доказује потпуно формално (нарочито не ручно)
- Постоје значајни примери формално верификованих програма (уз помоћ специјализованих доказивача, коришћењем специјализованих логика, попут Хорове логике):
  - верификовани оперативни системи
  - верификовани компилатори
  - верификован метро у Паризу
  - ...
- Техника инваријанти се може употребити и пре него што је програм написан у циљу извођења програмског кода из спецификације
- Техника инваријанти се може употребити и за откривање и исправљање грешака у датом програму

Пример: тробојка

Написати програм који учитава низ целих бројева а затим га трансформише тако да елементи буду подељени у три дела у зависности од задатих вредности A и B.

- У првом делу су елементи мањи од вредности A (вредности из интервала $[-\infty, A)$),
- у другом елементи већи или једнаки вредности A и мањи или једнаки вредности B (вредности из интервала $[A, B]$),
- у трећем елементи већи од вредности B (вредности из интервала $(B, +\infty)$).

Није битно у ком се редоследу налазе елементи унутар делова. Учитати елементе у низ, а затим реорганизовати редослед елемената у том низу (не користити помоћне низове).

https://petlja.org/biblioteka/r/zbirka-napredni-nivo/trobojka

Пример: бинарна претрага преломне тачке

Размотримо низ бројева 210, 2310, 390, 30, 510, 66, 6, 138, 46, 106, 59, 17, 23. Он је интересантан из неколико разлога.

- На пример, првих пет бројева је дељиво са 10, а после ниједан број није дељив са 10.
- Првих десет бројева је парно, а после су сви бројеви непарни.
- Првих осам бројева је дељиво са 6, а после ниједан број није дељив са 6.
- Прва два броја су дељива са 210, а после ниједан број није дељив са 210, итд.

Покушај да пронађеш још оваквих правилности. Напиши програм који за сваки унети делилац за који сигурно важи оваква правилност одређује колико бројева је дељиво са њим.

https://petlja.org/biblioteka/r/zbirka-napredni-nivo/prvi_paran

Пример: најмањи број који није збир елемената скупа

Дат је скуп природних бројева (задат у облику сортираног низа). Одредити најмањи природан број који није збир неких елемената тог скупа (сваки елемент скупа може само једном учествовати у збиру).

https://petlja.org/biblioteka/r/zbirka-napredni-nivo/najmanji_nezbir

Пример: превођење у бинарни запис

Доказати коректност алгоритма који преводи дати природни број у бинарни запис.

https://petlja.org/biblioteka/r/zbirka-napredni-nivo/binarni_zapis

Пример: Хорнерова схема

Написати програм којим се формира природан број од учитаних цифара, ако се цифре броја учитавају слева на десно (редом од цифре највеће тежине до цифре јединица) и доказати његову коректност.