

# CSCB039 Алгоритми и програмиране

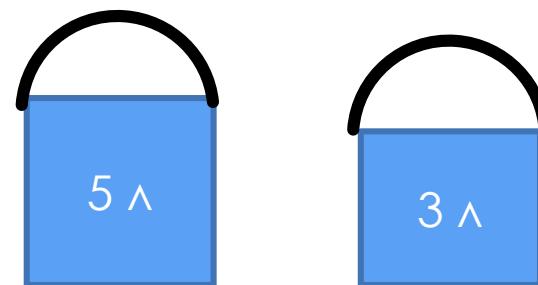
УВОДНА ЛЕКЦИОННА ЧАСТ

гл. ас. д-р Слав Емилов Ангелов, НБУ

# Задачка закачка

**Имаме две кофи с вода, едната е 3 л, а другата 5 л. Как да отмерим 4 литра вода с тези кофи?**

Можем да сипваме и отсипваме вода колкото пъти си искали.



# Изчислимост

**“Един математически проблем е изчислим, ако той по принцип може да се изчисли с компютърно устройство.”**

През 30-те са създадени няколко теории посветени на „изчислимост“:

- ▶ **Ламбда смятане** - Алонсо Чърч (Alonzo Church);
- ▶ **Рекурсивни функции** - Кърт Гьодел (Kurt Gödel);
- ▶ **Формални системи** – Стефан Клийни (Stephen Kleene);
- ▶ **Алгоритми на Марков** – Андрей Марков;
- ▶ **Машини на Тюинг (абстрактни машини)** – Алан Тюинг (Alan Turing) и Емил Пост (Emil Post).

**Изненадващо е, че изброените техники са еквивалентни!**

# Машини на Тюринг – формална дефиниция

Дефинирани за първи път през 1936 в статия “On Computable Numbers, with an Application to the Entscheidungsproblem”. Състоят се от:

- ▶ Крайно множество на възможните състояния  $Q$  (всяка такава машина в произволен момент трябва да бъде в едно състояние от  $Q$ );
- ▶ Памет - потенциално безкрайна лента, която съдържа последователни клетки  $\sigma_1, \sigma_2, \dots, \sigma_n$ . Те съдържат символи от крайна азбука  $\Sigma$  (виж следващия слайд);
- ▶ Глава на касетата, която може да чете и записва,  $h \geq 1$ , сканира клетка  $\sigma_h$ ;
- ▶ Функция на прехода,  $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{-1, 0, 1\}$ . Тази функция за всяко едно състояние  $q$ , сочещо към символ от клетка  $\sigma_h$ , назва, кое ще е новото състояние, новият символ, който трябва да бъде записан в съответната клетка и новата позиция на главата на касетата,  $h' = h + d$ . Където  $d$  е преместването, зададено от функцията на прехода  $\delta$ .

# Машини на Тюринг – неформално описание

Състоят се от:

- ▶ Памет – безкрайна лента, състояща се от клетки, във всяка от които е записан символ от никаква крайна азбука. Във всеки момент от работата на машината лентата е краина, но при нужда може да ѝ залепяме отляво или отдясно нови клетки, съдържащи празния символ.
- ▶ Глава, която във всеки момент от изчислението се намира над определена клетка от лентата. При всеки такт главата прочита символа от клетката, над която се намира, записва нов символ и се премества наляво или надясно по лентата в зависимост от изпълняваната инструкция и прочетения символ.
- ▶ Програма – краен списък от инструкции, който за разлика от съвременните компютри е отделен от паметта. Всяка инструкция е поредица от указания какво да се направи, ако главата е прочела i-тата буква от азбуката. Всяко указание съдържа информация какъв символ да се запише обратно върху лентата, коя инструкция ще се изпълнява на следващата стъпка и накъде (наляво или надясно) да се премести главата.
- ▶ Регистър, съдържащ номера на активната инструкция (програмен брояч). Една от инструкциите се приема за начална, т.е. изчислението започва със зареждането на номера ѝ в програмния брояч. Има и крайна инструкция – при достигането ѝ изчислението спира.

# Размисли

**Сравнете формалната и неформалната дефиниция.**

**Виждате ли, че са еквивалентни?**

# Един впечатляващ резултат

**През 1931, Гьодел шокира света с теоремата му за Непълнотата:**

**Няма пълно и изчислимо аксиоматизиране с логика от първи ред за естествените числа. Това означава, че няма смислено множество от аксиоми, с което да може да докажем всички верни твърдения от теория на числата.**

Какво е логика от първи ред?

Това е математически език, на който повечето математически твърдения могат да бъдат формулирани. Всяко твърдение, написано чрез логика от първи ред, има много прецизирано значение във всяка удачна логическа структура, т.е. то е вярно или грешно във всяка такава структура.

# Нерешими задачи

- ▶ Има ли алгоритъм, който да определя дали подадена на входа му компютърна програма решава задачата за удвояване на число, представено като поредица от единици (например  $11 \rightarrow 1111$ )?

**Теорема на Райс:** Няма алгоритъм, който да установява еквивалентност на компютърни програми.

**Следствие:** Не можем да създадем автоматизирани средства за проверка на коректността на компютърните програми

**Теорема (1936 г., Тюринг):** Няма алгоритъм, който да определя дали дадена програма ще завърши изчислението си или ще зацикли.

# СЛОЖНОСТ

Обикновено разглеждаме сложността като функция на време и/или памет.

Някои означения:

- **L** – логаритмично количество памет;
- **NL** – недетерминистично логаритмично пространство;
- **P** – полиномиално време;
- **PSPACE** – полиномно количество пространство на памет;
- **NP** – недетерминирано полиномиално време;
- **EXP** – експоненциално време;
- **NEXP** – недетерминирано експоненциално време;

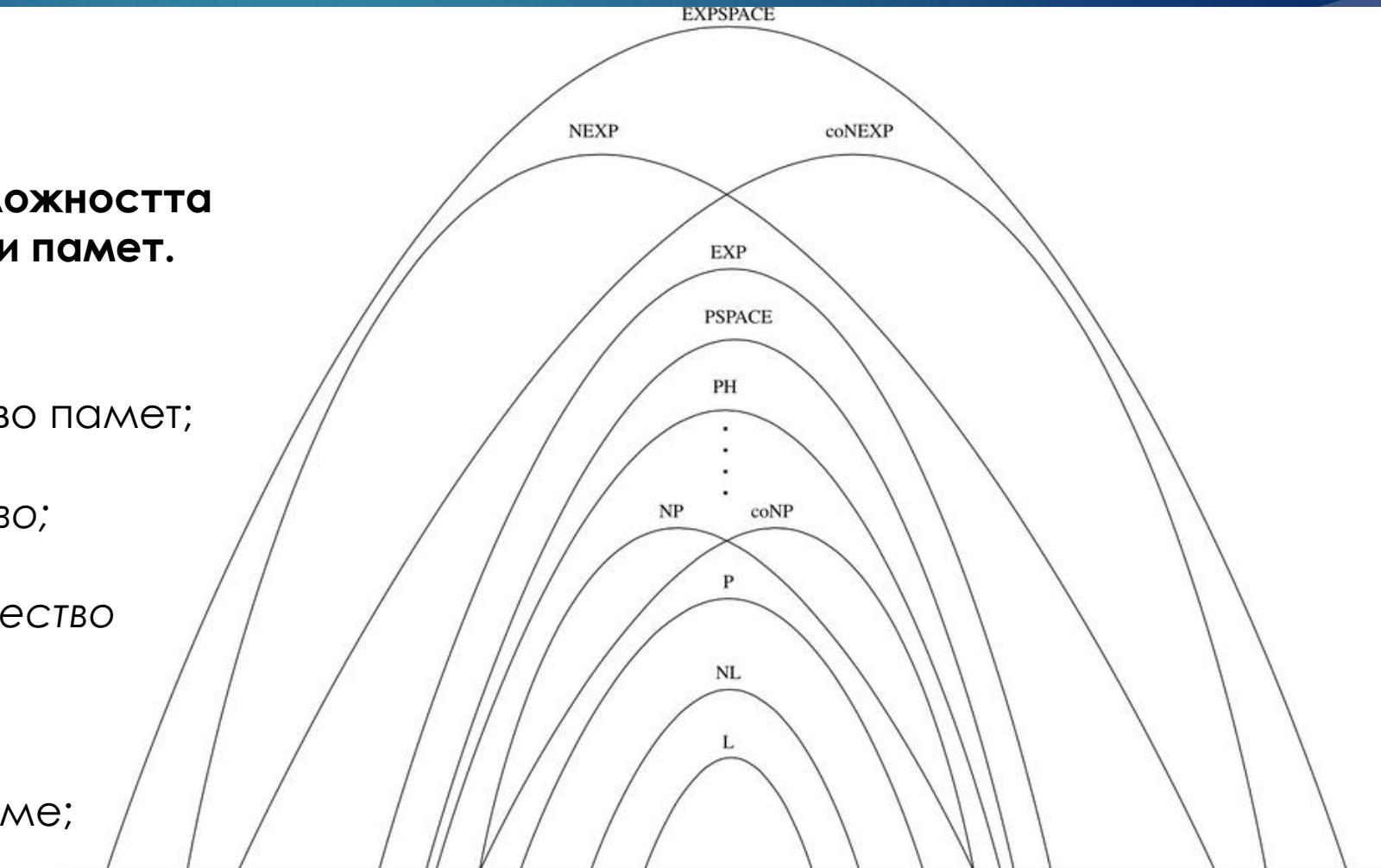
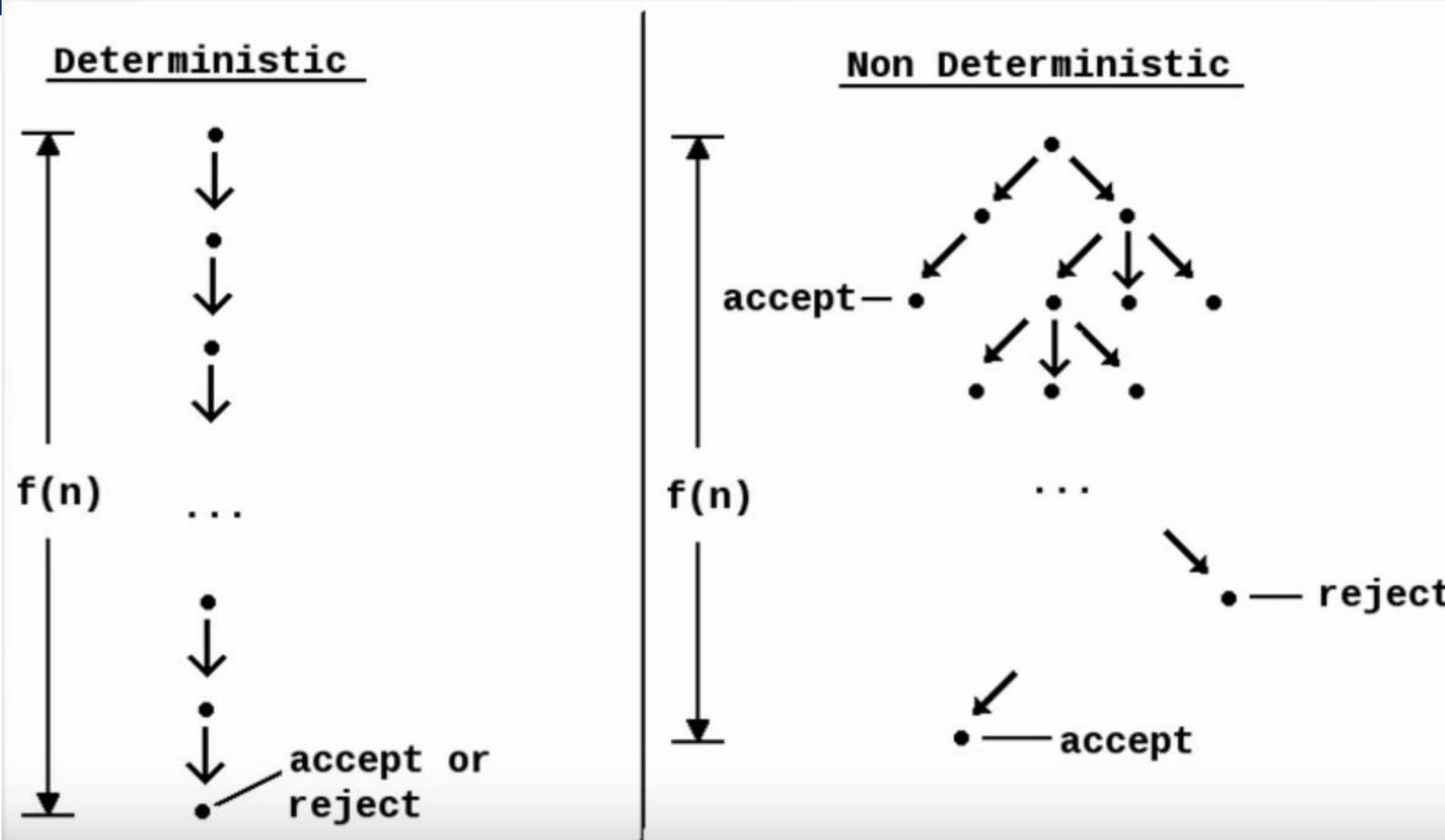


FIGURE 2. Inclusion relationships among major complexity classes. The only depicted inclusions which are currently known to be proper are  $\mathbf{L} \subsetneq \mathbf{PSPACE}$  and  $\mathbf{P} \subsetneq \mathbf{EXP}$ .

# Алгоритми според имплементацията

- ▶ **Рекурсивни или итеративни** - всяка рекурсивна версия на алгоритъм има еквивалентна (повече или по-малко сложна) итеративна версия, и обратно;
- ▶ **Логически** – вижте логическото програмиране;
- ▶ **Серийни, паралелни или разпределени**
- ▶ **Детерминирани и недетерминирани**
- ▶ **Точни или приблизителни**

# Детерминиран и недетерминиран



Още по темата:  
<https://www.youtube.com/watch?v=iJmMw0MpaKY>

# Алгоритми според дизайна

- ▶ „Груба сила“ – чрез изчерпване на случаите;
- ▶ „Разделяй и владей“
- ▶ Динамично програмиране
- ▶ Линейно програмиране
- ▶ Вероятностни и евристични алгоритми

# Целочислени редици - формули

- Частични суми на естествените числа:

$$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}.$$

- Частични суми на квадратите им:

$$1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

- Други (връзка с квадратите на триъгълните числа):

$$1^3 + 2^3 + 3^3 + \dots + n^3 = \frac{n^2(n+1)^2}{4} \text{ или}$$

$$1^3 + 2^3 + 3^3 + \dots + n^3 = (1 + 2 + 3 + \dots + n)^2.$$

**Защо могат да бъдат  
важни за  
програмирането?**

# Прогресии

- ▶ Аритметична:  $a_1, a_1 + d, a_1 + 2d, a_1 + 3d, \dots$

$$a_n = a_1 + (n - 1)d, \quad \sum_{i=1}^n a_n = \frac{n(a_1 + a_n)}{2}.$$

- ▶ Геометрична:  $a_1, da_1, d^2a_1, d^3a_1, d^4a_1, \dots$

$$a_n = a_1 d^{n-1}, \quad \sum_{i=1}^n a_n = \frac{a_1(1-d^n)}{1-d}.$$

# CSCB039 Алгоритми и програмиране

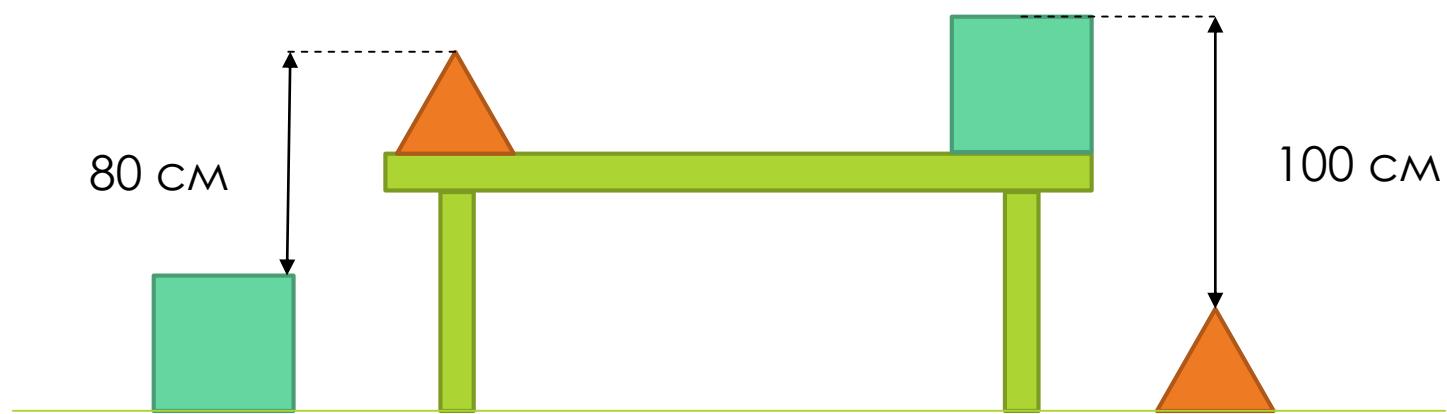
ЛЕКЦИЯ 2

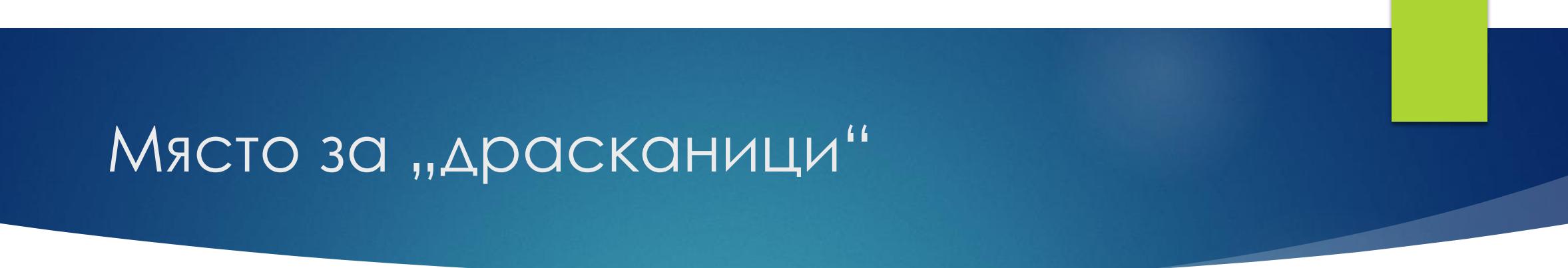
гл. ас. д-р Слав Емилов Ангелов, НБУ

# Загрявка

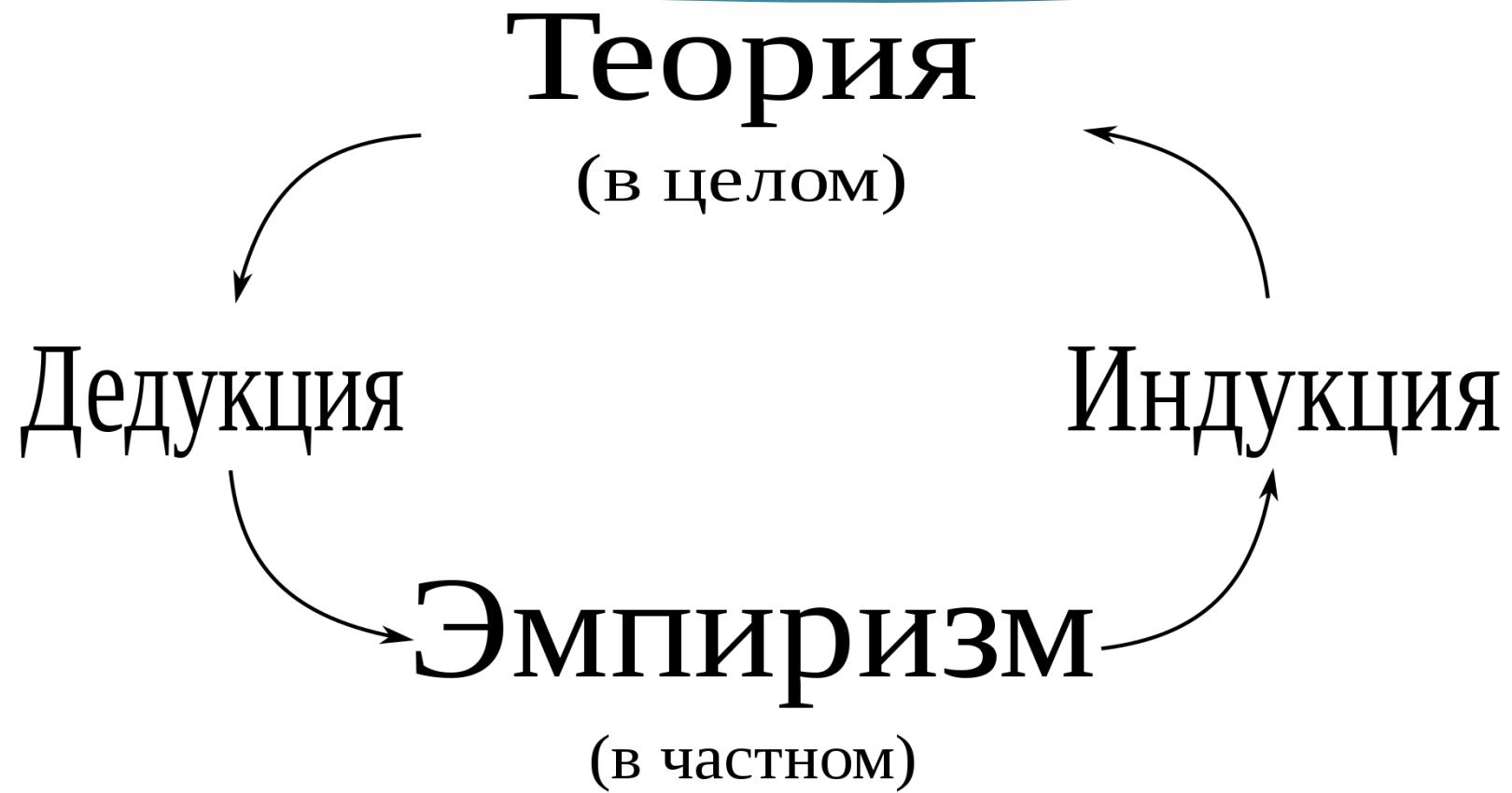
Задача за олимпиада  
по математика за 4  
клас в Китай.

**Колко е височината  
на масата ?**





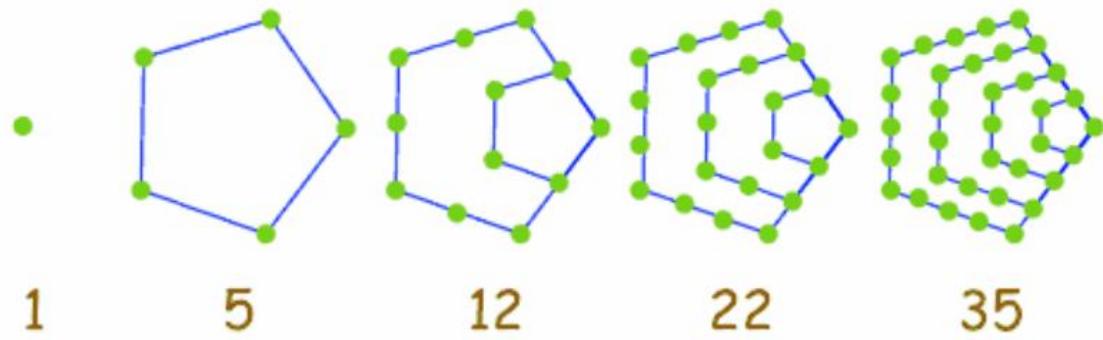
Място за „драсканици“



# Петоъгълни числа

Начални членове: 1, 5, 12, 22, 35, 51, 70, 92, 117, 145, 176, 210, 247, 287, 330, 376, ....

$$p_n = \frac{3n(n - 1)}{2} = C(n, 1) + 3C(n, 2).$$



Връзки с триъгълните числа:

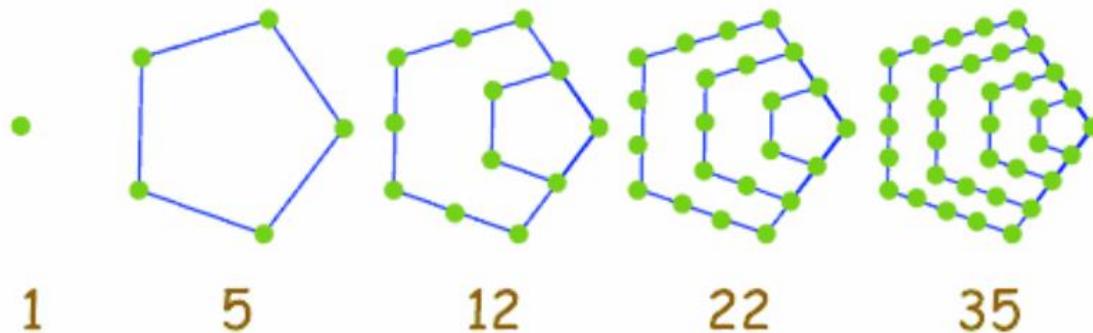
# Петоъгълни числа

Начални членове: 1, 5, 12, 22, 35, 51, 70, 92, 117, 145, 176, 210, 247, 287, 330, 376, ....

$$p_n = \frac{3n(n - 1)}{2} = C(n, 1) + 3C(n, 2).$$

Тест:  $n = \frac{\sqrt{24x+1}+1}{6}$ ,  $n$  естествено число.

Връзки с триъгълните числа:

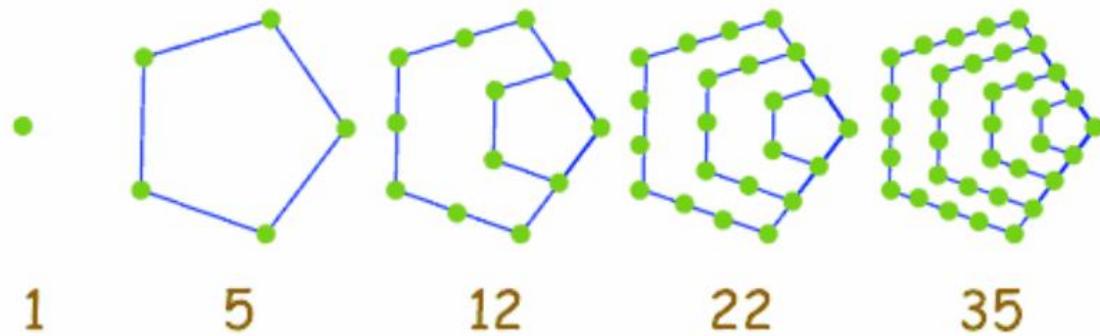


# Петоъгълни числа

Начални членове: 1, 5, 12, 22, 35, 51, 70, 92, 117, 145, 176, 210, 247, 287, 330, 376, ....

$$p_n = \frac{3n(n - 1)}{2} = C(n, 1) + 3C(n, 2).$$

Тест:  $n = \frac{\sqrt{24x+1}+1}{6}$ ,  $n$  естествено число.



Връзки с триъгълните числа:

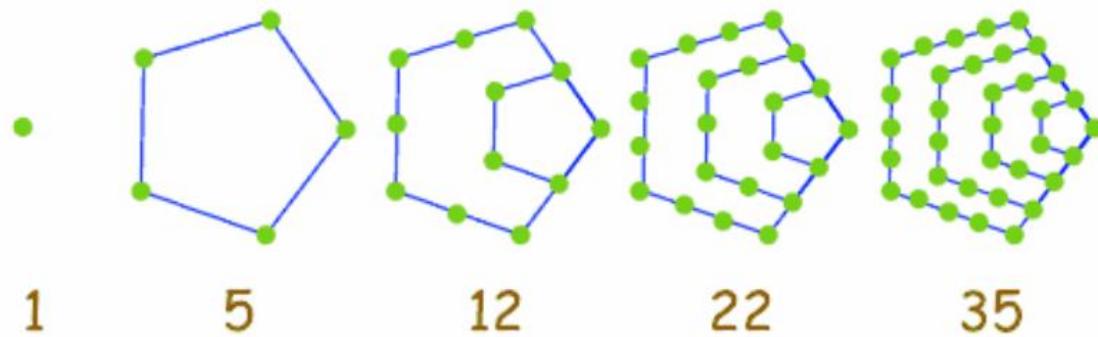
- ▶ n-тото петоъгълно число  $p_n$  е равно на една трета от  $(3n-1)$ -тото триъгълно  $T_{3n-1}$ ;

# Петоъгълни числа

Начални членове: 1, 5, 12, 22, 35, 51, 70, 92, 117, 145, 176, 210, 247, 287, 330, 376, ....

$$p_n = \frac{3n(n-1)}{2} = C(n, 1) + 3C(n, 2).$$

Тест:  $n = \frac{\sqrt{24x+1}+1}{6}$ ,  $n$  естествено число.



Връзки с триъгълните числа:

- ▶ n-тото петоъгълно число  $p_n$  е равно на една трета от  $(3n-1)$ -тото триъгълно  $T_{3n-1}$ ;
- ▶  $p_n = T_{n-1} + n^2$  (какво виждаме за всеки квадрат на естествено число?);

# Петоъгълни числа

Начални членове: 1, 5, 12, 22, 35, 51, 70, 92, 117, 145, 176, 210, 247, 287, 330, 376, ....

$$p_n = \frac{3n(n-1)}{2} = C(n, 1) + 3C(n, 2).$$

**Тест:**  $n = \frac{\sqrt{24x+1}+1}{6}$ ,  $n$  е естествено число.

Начални членове: 1, 5, 12, 22, 35, 51, 70, 92, 117, 145, 176, 210, 247, 287, 330, 376, ....

Тест:  $n = \frac{\sqrt{24x+1}+1}{6}$ ,  $n$  е естествено число.

## Връзки с триъгълните числа:

- ▶ n-тото петоъгълно число  $p_n$  е равно на една трета от  $(3n-1)$ -тото триъгълно  $T_{3n-1}$ ;
- ▶  $p_n = T_{n-1} + n^2$  (какво виждаме за всеки квадрат на естествено число?);
- ▶  $p_n = T_n + 2T_{n-1}$  (сега може ли да изразим  $n^2$  само с триъгълни числа?);
- ▶  $p_n = T_{2n-1} - T_{n-1}$ .

## Връзки с триъгълните числа:

- ▶ n-тото петоъгълно число  $p_n$  е равно на една трета от  $(3n-1)$ -тото триъгълно  $T_{3n-1}$ ;
- ▶  $p_n = T_{n-1} + n^2$  (какво виждаме за всеки квадрат на естествено число?);
- ▶  $p_n = T_n + 2T_{n-1}$  (сега може ли да изразим  $n^2$  само с триъгълни числа?);
- ▶  $p_n = T_{2n-1} - T_{n-1}$ .

# The halting problem

**Понеже машините на Тюринг са конструирани да завършат всички възможни изчисления, то понякога се случва да не могат да спрат.**

Примери:

# The halting problem

**Понеже машините на Тюринг са конструирани да завършат всички възможни изчисления, то понякога се случва да не могат да спрат.**

Примери:

1. На 18-то състояние може да гледа към 1, после да запише отново 1 и да отиде отново в 18 с нулево изменение (тривиален пример);

# The halting problem

**Понеже машините на Тюринг са конструирани да завършат всички възможни изчисления, то понякога се случва да не могат да спрат.**

Примери:

1. На 18-то състояние може да гледа към 1, после да запише отново 1 и да отиде отново в 18 с нулево изменение (тривиален пример);
2. Може да достигнем до празен символ, а на дясно да са само празни символи. После тръгваме на дясно да търсим 1 (малко по-адекватен);

# The halting problem

**Понеже машините на Тюринг са конструирани да завършат всички възможни изчисления, то понякога се случва да не могат да спрат. Видео:**

<https://www.youtube.com/watch?v=92WHN-pAFCs>

Примери:

1. На 18-то състояние може да гледа към 1, после да запише отново 1 и да отиде отново в 18 с нулево изменение (тривиален пример);
2. Може да достигнем до празен символ, а на дясно да са само празни символи. После тръгваме на дясно да търсим 1 (малко по-адекватен);
3. Представете си машина на Тюринг, която на определен вход започва да търси първият контрапример на Голямата теорема на Ферма. От сравнително скоро знаем, че машината ще го търси до безкрай....

**Това да изпринтираме в десетичен вид 1/3 към кой от примерите приспада?**

# Голямата теорема на Ферма

**Теорема:**

**НЕ съществуват цели положителни числа  $x$ ,  $y$  и  $z$ , за които да е изпълнено  $x^n + y^n = z^n$  при  $n > 2$ .**

**Има ли множество решения за  $n=2$  ?**

Теоремата за първи път е доказана през 1994 от **Андрю Уайлс**, но с грешка. Той коригира грешката две години по-късно, като трудът му е 150 страници. Уайлс за постижението си получава **Абелова награда** през 2016.

# Тезис на Чърч

Вече споменахме, че техниките създадени през 30-те за установяване на ефективност на методи (формални процедури, алгоритми, изчислителни машини, компютри) са доказано евивалентни. **Алонсо Чърч изказва твърдението, че това е като природен закон за всички възможни схеми за изчисляване.**

**Следствие:** Не е необходимо, когато решаваме дадена задача да се мъчим да конструираме машина на Тюринг. Може просто да я напишем на удобен за нас език за програмиране. Ако успеем, това гарантира, че има реализация и с машина на Тюринг.

# При приемане тезиса на Чърч

**Защо има толкова голямо множество езици за програмиране при положение, че те са еквивалентни по отношение на Изчислимост ? Защо не един?**

# При приемане тезиса на Чърч

**Защо има толкова голямо множество езици за програмиране при положение, че те са еквивалентни по отношение на Изчислимост ? Защо не един?**

Възможен отговор: Може да разглеждаме всеки език като съставно на две части – математическа (идейна) и човешка (реализация). Математическата част се състои от строго дефинирани модели, които са ефективни и с възможни разлики в ефикасността. До тук добре, но човешката част е нещо, което е в страни от изчислимост и сложност. Тя включва елементи като специализация на езика, вкусови предпочитания към синтаксис и семантика, хронология на възникване на езиците, разпространение и др..

# Алгоритмично нерешими задачи

- ▶ Всеки път, когато пуснем машина да търси решение на задача, за която от теорията се знае, че няма такова (вижте Голямата теорема на Ферма);
- ▶ Машина, която да търси първото нечетно **съвършено число** (все още не се знае дали има такова);
- ▶ Много други задачи, но изискват прекалено много разяснения, че да ги формулираме ясно в този курс...

# CSCB039 Алгоритми и програмиране

ЛЕКЦИЯ 3

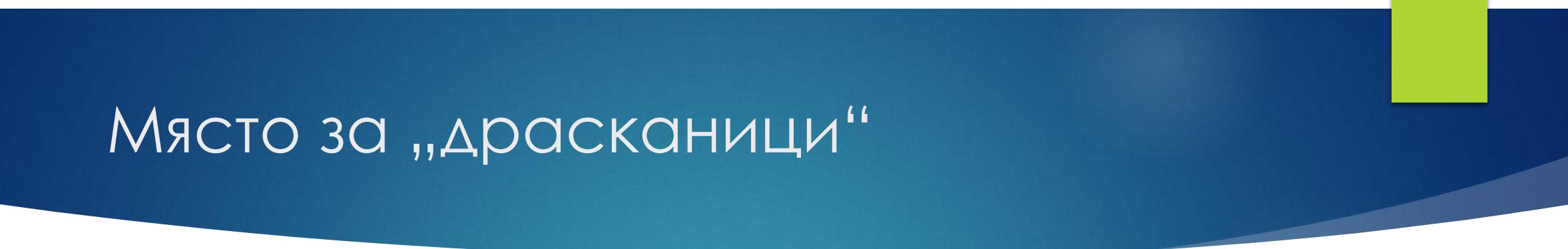
гл. ас. д-р Слав Емилов Ангелов, НБУ

# Загрявка

Знаем, че в С и С++ индексацията започва от индекс 0. Ами, ако много искаме да започва от 1, използваме следната техника ( $n$  е дължината на масива):



- Плюсове?
- Минуси?



Място за „драсканици“

# Сложна лихва – извеждане на формула

Нека имаме парична сума  $P$ , която се олихвява със сложна лихва от  $r$  процента в продължение на  $T$  периода. Да се опитаме да спестим итеративното пресмятане. Да разпишем какво получаваме като лихва от всеки един период:

**Период 1**

$$P \cdot r$$

$$\downarrow$$
  
$$P \cdot r$$

**Период 2**

$$(P + P \cdot r)r$$

$$\downarrow$$
  
$$P(1 + r)r$$

**Период 3**

$$[P + P \cdot r + P(1 + r)r]r$$

$$\downarrow$$
  
$$[P(1 + r) + P(1 + r)r]r$$

$$P(1 + r)(1 + r)r$$

**Период 4**

$$[P + P \cdot r + P(1 + r)r + P(1 + r)^2r]r$$

$$\downarrow$$
  
$$[P(1 + r)^2 + P(1 + r)^2r]r$$

$$P(1 + r)^2(1 + r)r$$

**Период T** (по индукция):  $P(1 + r)^{T-1}r$

**Сега да съберем всички вземания.**

# Окончателна формула

**Можем ли да намерим подход за кратка формула за сумата:**

$$S_T = P + P \cdot r + P(1 + r)r + P(1 + r)^2r + P(1 + r)^3r + P(1 + r)^4r + \cdots + P(1 + r)^{T-1}r$$

?

# Окончателна формула

**Можем ли да намерим подход за кратка формула за сумата:**

$$S_T = P + P \cdot r + P(1 + r)r + P(1 + r)^2r + P(1 + r)^3r + P(1 + r)^4r + \cdots + P(1 + r)^{T-1}r$$

Геометрична прогресия,  $S_n = \frac{a(1-q^n)}{(1-q)}$  ?

# Окончателна формула

Можем ли да намерим подход за кратка формула за сумата:

$$S_T = P \cdot r + P(1+r)r + P(1+r)^2r + P(1+r)^3r + P(1+r)^4r + \cdots + P(1+r)^{T-1}r$$

Геометрична прогресия,  $S_n = \frac{a(1-q^n)}{(1-q)}$  ?

В нашия случай имаме  $a = P \cdot r$ , а  $q = 1 + r$ . Тогава:

Приключихме  
ли?

$$S_T = \frac{P \cdot r[1 - (1+r)^T]}{1 - (1+r)} = \frac{P \cdot r[1 - (1+r)^T]}{-r} = \frac{P \cdot r[(1+r)^T - 1]}{r} = P[(1+r)^T - 1]$$

# Окончателна формула

Можем ли да намерим подход за кратка формула за сумата:

$$S_T = P \cdot r + P(1+r)r + P(1+r)^2r + P(1+r)^3r + P(1+r)^4r + \cdots + P(1+r)^{T-1}r$$

Геометрична прогресия,  $S_n = \frac{a(1-q^n)}{(1-q)}$  ?

В нашия случай имаме  $a = P \cdot r$ , а  $q = 1 + r$ . Тогава:

$$S_T = \frac{P \cdot r[1 - (1+r)^T]}{1 - (1+r)} = \frac{P \cdot r[1 - (1+r)^T]}{-r} = \frac{P \cdot r[(1+r)^T - 1]}{r} = P[(1+r)^T - 1]$$

Приключихме

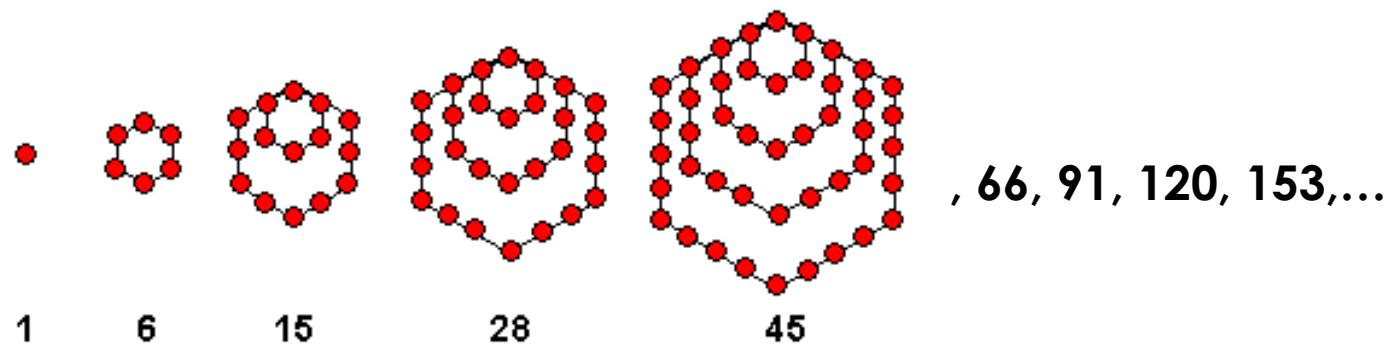
ли?

Ами  
първоначалната  
сума  $P$ ?

# Шестоъгълни числа

Първите шестоъгълни числа са:

Тест:  $n = \frac{\sqrt{8x + 1} + 1}{4}$ .



Формули за общий член:  $h_n = 2n^2 - n$ .

Интересни факти:

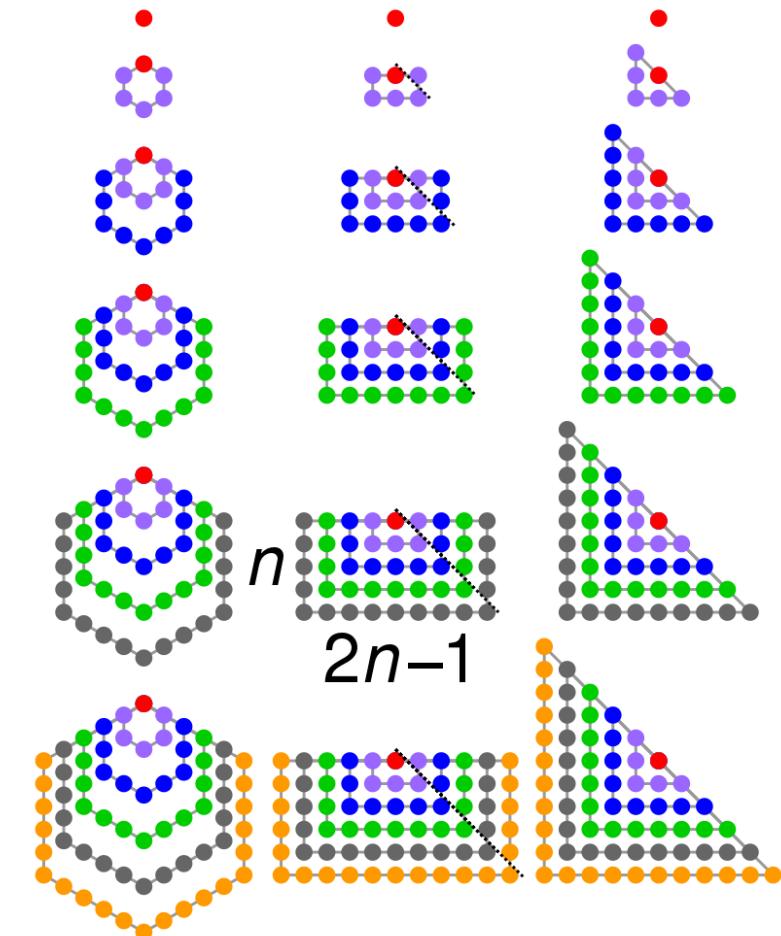
- Всяко четно съвършено число е шестоъгълно число;
- Най-голямото цяло число, което не може да бъде изразено като сума на не повече от 4 шестоъгълни числа е 130;
- Често под шестоъгълни числа се подразбират „центрираните шестоъгълни числа“, което е друго.

# Шестоъгълни числа – връзка с правоъгълните и триъгълните числа

**Може ли да определите по схемата  $n$ -тото шестоъгълно число на кое правоъгълно отговаря, а на кое триъгълно ?**

Забележка:

Това, което тук се цитира като правоъгълно число НЕ отговаря на правоъгълните числа (за тях ще говорим другия път).



# Шестоъгълни числа – връзка с простите числа на Мерсен

Мерсеново число  $M_p$  е всяко число, което се получава по формулата  $2^n - 1$ ,  $n > 1$ . Малко от получените такива числа са прости и се наричат **мерсенови прости числа**. **Всяко мерсеново число, което образува съвършено число е мерсеново просто число.**

На лице са следните зависимости, свързващи  $M_p$  с шестоъгълните числа и съвършените числа:

$$\text{съвършено число} = 2^{p-1}M_p = M_p \frac{M_p + 1}{2} = h_{\frac{M_p+1}{2}} = h_{2^{p-1}}, p = 2, 3, 5, 7, 13, 19, 31, \dots, \text{A000043}$$

Първите **мерсенови числа**: 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047, 4095, 8191, 16383, ...

Първите **мерсенови прости числа**: 3, 7, 31, 127, 8191, 131 071, 524 287, 214 748 3647, ...

Първите **съвършени числа**: 6, 28, 496, 8128, 33 550 336, 8 589 869 056, 137 438 691 328, ...

Първите **шестоъгълни числа**: 1, 6, 15, 28, 45, 66, 91, 120, 153, 190, 231, 276, 325, 378, 435, ...

# Сложност – базови понятия

**Бързодействието на даден алгоритъм зависи от машината, на която се изпълнява и големината на входа.**

Най-важният аспект в теорията за сложност на алгоритъм е асимптотичната сложност на алгоритъм в най-лошите случаи, като функция на дълчината на входа  $n$ .

**Дефиниция:** Казваме, че **машина на Тюринг  $M$  се изпълнява за време  $T(n)$** , ако за всеки вход  $w$  с дължина  $n$ ,  $M(w)$  отнема най-много  $T(n)$  стъпки и спира.

# БАЗОВИ ПОНЯТИЯ (тежка формализация)

При сложност ще говорим за машини на Тюринг, които се изпълняват до край при всеки зададен ход. Нека да приемем, че дадена машина на Тюринг връща 1, ако може да изпълни подаден вход, 0, ако ще зацикли. Тогава може да дефинираме множеството от допустимите входове:

$$L(M) = \{w \mid M(w) = 1\}.$$

Разглеждаме функция  $T(n)$  от предходния слайд. За всяка функция  $T: N \rightarrow N$  дефинираме:

$$Time[T(n)] = \{A \mid A = L(M), \quad M \text{ се изпълнява за време } T(n)\}.$$

**Сега може да дефинираме класът на сложност P като:**

$$P = \bigcup_{i=1,2,3\dots} Time[n^i].$$

# Ниски сложности - примери

- ▶ **Логаритмична** – например, търсенето на елемент в сортиран масив с двоично търсене;
- ▶ **Линейна** – обхождането на масив;
- ▶ **Линейна** – сортиране на числата от 1 до  $n$ , които са в обратен ред;
- ▶ **Линейна** – сортиране на разбърканите числа от 1 до  $n$ .



**Това не е ли логаритмична сложност ?**

# Още за класовете на сложност

- ▶ Задачите от **клас на сложност P** са решими дори и за сравнително големи дължини на входа. Обикновено за задачите от практиката с такава сложност могат да се създадат алгоритми за решаването им.

Други класове на сложност:

- ▶ **PSPACE** – включва проблеми решими със заделяне на полиномно количество пространство на памет;
- ▶ **EXPTIME** – включва проблемите решими за време  $2^{p(n)}$ ,  $p(n)$  е даден полином;
- ▶ **NP** – включва проблемите, изискващи недетерминирано полиномно време.

# Сводимост (Reduction)

Искаме да сравним относителната трудност на два проблема A и B. Казваме, че **A се свежда до B, ако съществува лесна за изчисляване трансформация  $\tau$ , която свързва елементите на A и B така, че  $\tau(w) \in B \leftrightarrow w \in A$ .**

Самата функция  $\tau$ , която свежда един проблем до друг може да свежда за полиномиално време. Например Карп [Karp] използва такива, за да покаже, че редица важни комбинаторни проблеми са NP пълни. Известни са редица функции  $\tau$  – редукции отнемащи логаритмично количество памет, такива за първи ред логика, проекции, първи ред проекции.

# Пълнота

**Проблем A е пълен за клас на сложност C**, ако A е в C и всички други проблеми B от C не са по-трудни от A, т.е.  $A \leq B$ .

Интересно е, че много естествено възникващи проблеми се оказват пълни по отношение на някой от вече споменатите класове на сложност. Напълно точно обяснение на този феномен все още не е известно.

# Задачата за осемте царици

**По колко начина можем да разположим 8 царици на шахматната дъска така, че те да не се застрашават една с друга?**

Днес има 92 решения, а възможните разположения на дамите са близо 4 милиарда и половина. Според информация от 2017 година, наградата за намиране на ефикасен алгоритъм е 1 million долара!

Източник: <https://nova.bg/news/view/2017/09/07/192333/%D0%B7%D0%B0%D0%B4%D0%B0%D1%87%D0%B0-%D0%B7%D0%B0-1-%D0%BC%D0%B8%D0%BB%D0%B8%D0%BE%D0%BD-%D0%B4%D0%BE%D0%BB%D0%B0%D1%80%D0%B0-%D0%B2%D0%BB%D0%B8%D0%B4%D0%B5%D0%BE/>

# Продължение на лекцията

**Ще продължим с презентация от проф. Манев за асимптотично поведение на функция.**

**От дясно прилагам как се произнасят гръцките букви (живично важно в математиката).**

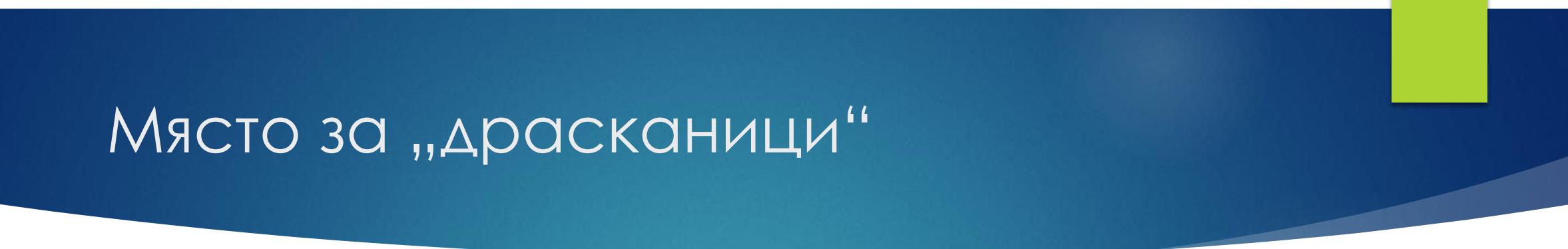
|       |         |         |       |         |       |
|-------|---------|---------|-------|---------|-------|
| A     | B       | Г       | Δ     | Е       | Z     |
| Alpha | Beta    | Gamma   | Delta | Epsilon | Zeta  |
| Η     | Θ       | I       | Κ     | Λ       | Μ     |
| Eta   | Theta   | Iota    | Kappa | Lambda  | Mu    |
| N     | Ξ       | Ο       | Π     | Ρ       | Σ     |
| Nu    | Xi      | Omicron | Pi    | Rho     | Sigma |
| Τ     | Υ       | Φ       | Χ     | Ψ       | Ω     |
| Tau   | Upsilon | Phi     | Chi   | Psi     | Omega |

|       |         |         |       |         |       |
|-------|---------|---------|-------|---------|-------|
| α     | β       | γ       | δ     | ε       | ζ     |
| Alpha | Beta    | Gamma   | Delta | Epsilon | Zeta  |
| η     | θ       | ι       | κ     | λ       | μ     |
| Eta   | Theta   | Iota    | Kappa | Lambda  | Mu    |
| ν     | ξ       | ο       | π     | ρ       | σ     |
| Nu    | Xi      | Omicron | Pi    | Rho     | Sigma |
| τ     | υ       | φ       | χ     | ψ       | ω     |
| Tau   | Upsilon | Phi     | Chi   | Psi     | Omega |

# CSCB039 Алгоритми и програмиране

ЛЕКЦИЯ 4

гл. ас. д-р Слав Емилов Ангелов, НБУ



Място за „драсканици“

# Центрирани шестоъгълни числа

Първи членове: 1, 7, 19, 37, 61, 91, 127, 169, 217, 271, ...

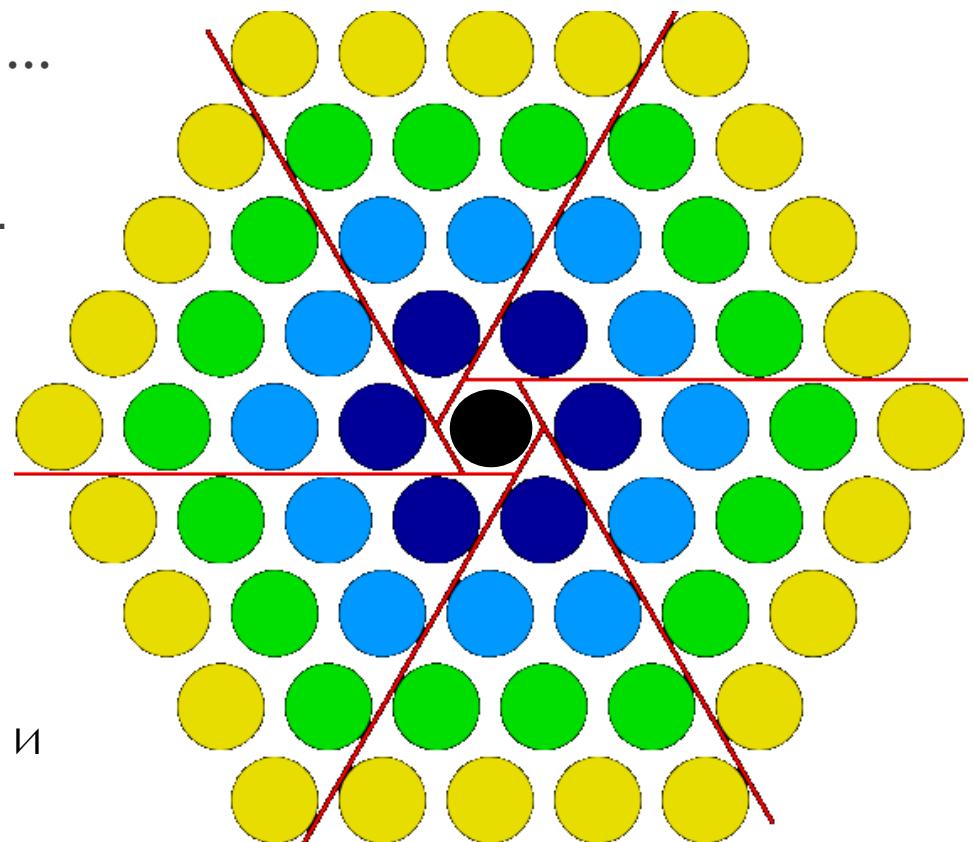
Формула:

$$H(n) = n^3 - (n - 1)^3 = 3n(n - 1) + 1 = 3n^2 - 3n + 1.$$

**От формулите се вижда връзката с кубичните, триъгълните и правоъгълните числа.**

Допълнително:

- ▶  $\sum_{i=1}^n H(i) = n^3$  - n-тото кубично число е сума на първите n шестоъгълни, т.е. кубичните числа са равни на шестоъгълните пирамидални числа;
- ▶  $H(n) = (2n - 1)^2 - n(n - 1)$  – връзката между  $H(n)$  и  $(n-1)$ -то правоъгълно число  $n(n-1)$ .



# Продължение на лекцията

**Ще продължим с материал от проф. Манев за асимптотично поведение на функция.**

**От дясно прилагам как се произнасят гръцките букви (живично важно в математиката).**

|       |         |         |       |         |       |
|-------|---------|---------|-------|---------|-------|
| A     | B       | Г       | Δ     | Е       | Z     |
| Alpha | Beta    | Gamma   | Delta | Epsilon | Zeta  |
| Η     | Θ       | I       | Κ     | Λ       | Μ     |
| Eta   | Theta   | Iota    | Kappa | Lambda  | Mu    |
| N     | Ξ       | Ο       | Π     | Ρ       | Σ     |
| Nu    | Xi      | Omicron | Pi    | Rho     | Sigma |
| Τ     | Υ       | Φ       | Χ     | Ψ       | Ω     |
| Tau   | Upsilon | Phi     | Chi   | Psi     | Omega |

|       |         |         |       |         |       |
|-------|---------|---------|-------|---------|-------|
| α     | β       | γ       | δ     | ε       | ζ     |
| Alpha | Beta    | Gamma   | Delta | Epsilon | Zeta  |
| η     | θ       | ι       | κ     | λ       | μ     |
| Eta   | Theta   | Iota    | Kappa | Lambda  | Mu    |
| ν     | ξ       | ο       | π     | ρ       | σ     |
| Nu    | Xi      | Omicron | Pi    | Rho     | Sigma |
| τ     | υ       | φ       | χ     | ψ       | ω     |
| Tau   | Upsilon | Phi     | Chi   | Psi     | Omega |

# $O(g(n))$

Нека с  $I_1$  означим множеството на функциите на една неотрицателна цяла променлива.

**Дефиниция 1:** Казваме, че  $f(n)$  принадлежи на множеството  $O(g(n))$ , ако **съществуват положителни константи  $c$  и  $n_0$**  такива, че  $0 \leq f(n) \leq c.g(n), \forall n \geq n_0$

Вместо  $f(n) \in O(g(n))$  по-често пишем  $f(n) = O(g(n))$

# $O(g(n))$

Нека с  $I_1$  означим множеството на функциите на една неотрицателна цяла променлива.

**Дефиниция 1:** Казваме, че  $f(n)$  принадлежи на множеството  $O(g(n))$ , ако **съществуват положителни константи  $c$  и  $n_0$**  такива, че  $0 \leq f(n) \leq c.g(n), \forall n \geq n_0$

Вместо  $f(n) \in O(g(n))$  по-често пишем  $f(n) = O(g(n))$

**Пример:**  $n^2 \in O(2n^2)$ . Защо? Очевидно е, че при  $c = 1$  и  $n_0 = 0$  имаме  $0 \leq n^2 \leq 1.2n^2, \forall n \geq 0$ .

# $O(g(n))$ - описание

Нека с  $I_1$  означим множеството на функциите на една неотрицателна цяла променлива.

**Дефиниция 1:** Казваме, че  $f(n)$  принадлежи на множеството  $O(g(n))$ , ако **съществуват положителни константи  $c$  и  $n_0$**  такива, че  $0 \leq f(n) \leq c.g(n)$ ,  $\forall n \geq n_0$

Вместо  $f(n) \in O(g(n))$  по-често пишем  $f(n) = O(g(n))$

**Пример:**  $n^2 \in O(2n^2)$ . Защо? Очевидно е, че при  $c = 1$  и  $n_0 = 0$  имаме  $0 \leq n^2 \leq 1.2n^2$ ,  $\forall n \geq 0$ .

**Лема 1а.**  $\forall f(n) \in I_1 \in$  в сила  $f(n) = O(f(n))$ .

# $O(g(n))$ - описание

Нека с  $I_1$  означим множеството на функциите на една неотрицателна цяла променлива.

**Дефиниция 1:** Казваме, че  $f(n)$  принадлежи на множеството  $O(g(n))$ , ако **съществуват положителни константи  $c$  и  $n_0$**  такива, че  $0 \leq f(n) \leq c.g(n)$ ,  $\forall n \geq n_0$

Вместо  $f(n) \in O(g(n))$  по-често пишем  $f(n) = O(g(n))$

**Пример:**  $n^2 \in O(2n^2)$ . Защо? Очевидно е, че при  $c = 1$  и  $n_0 = 0$  имаме  $0 \leq n^2 \leq 1.2n^2$ ,  $\forall n \geq 0$ .

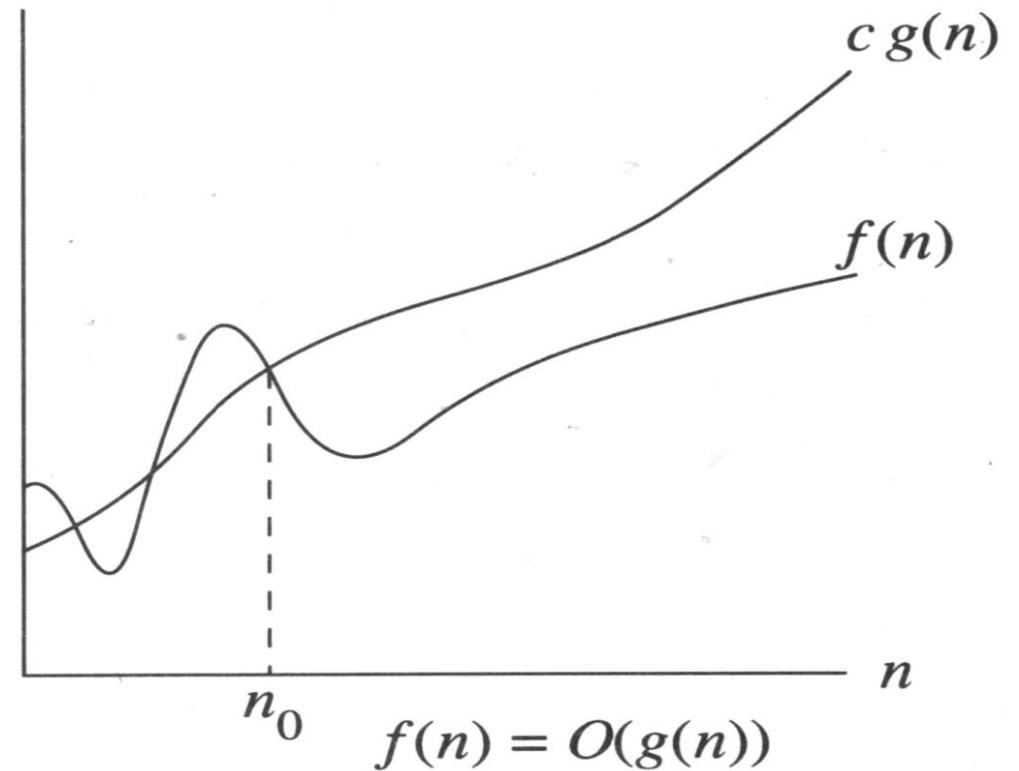
**Лема 1а.**  $\forall f(n) \in I_1$  е в сила  $f(n) = O(f(n))$ .

**Доказателство:** за  $c = 1$ ,  $n_0 = 0$  имаме

$$0 \leq f(n) \leq 1 \cdot f(n), \forall n \geq 0$$

# $O(g(n))$ - онагледяване

С прости думи  $f(n) \in O(g(n))$  означава че от някакво място нататък, до безкрайност, графиката на  $f(n)$  остава под графиката на  $c.g(n)$ .



# $O(g(n))$ – важен пример

**Неочевиден пример:**  $2n^2+13 \in O(n^2)!$

Доказательство:

# $O(g(n))$ – важен пример

**Неочевиден пример:**  $2n^2+13 \in O(n^2)!$

Доказателство:

Търсим  $c$  и  $n_0$  такива, че  $2n^2+13 \leq c.n^2$ ,  $\forall n \geq n_0$ . Очевидно  $c=2$  не върши работа, защото  $2n^2+13 \leq 2n^2$  не е в сила за никое  $n$ .

Затова нека  $c=3$ .

Кога  $2n^2+13 \leq 3n^2$  ? При  $n = 3$  имаме  $2n^2+13=31$ , а  $3n^2=27$ .

Но при  $n = 4$ :  $2n^2+13=45$ , а  $3n^2=48$ . И за всяко  $n > 4$  неравенството също е в сила.  
Значи

$$2n^2+13 \leq 3.n^2, \forall n \geq n_0 = 4.$$

## $o(g(n))$ - описание

**Дефиниция 2:** Функцията  $f(n)$  принадлежи на множеството  $o(g(n))$ , ако за **Всички положителни константи съществува**  $n_0$  такова, че  $0 \leq f(n) \leq c.g(n)$ ,  $\forall n \geq n_0$

Пример:

# $O(g(n))$ - описание

**Дефиниция 2:** Функцията  $f(n)$  принадлежи на множеството  $O(g(n))$ , ако за **Всички положителни константи съществува**  $n_0$  такова, че  $0 \leq f(n) \leq c.g(n)$ ,  $\forall n \geq n_0$

Пример:

Функцията  $n \in O(n^2)$ . Защо? Очевидно е, че при  $c = 1$  и  $n_0 = 0$  имаме  $0 \leq n \leq 1 \cdot n^2$ ,  $\forall n \geq 0$

Дали  $n \leq cn^2$  за всяка константа  $c > 0$ ? Кога  $n > cn^2$  – когато  $1 > cn$  или  $n < 1/c$ . Значи  $\forall c > 0$  и  $n \geq 1/c = n_0$  ще е в сила  $0 \leq n \leq cn^2$

# $O(g(n))$ – някои леми

**Лема 16.** За всеки  $p > q > 0$  е в сила  $n^q = o(n^p)$ .

**Доказателство:**

# $O(g(n))$ – някои леми

**Лема 16.** За всеки  $p > q > 0$  е в сила  $n^q = o(n^p)$ .

**Доказателство:**

Нека  $c > 0$ . Да потърсим  $n_0$  такова, че  $0 \leq n^q \leq c \cdot n^p, \forall n \geq n_0$ , т.е.  $1 \leq c \cdot n^{p-q}$  и значи  $n_0$  трябва да е  $\geq$  от  $(p-q)$ -ти корен от  $1/c$ .

# $O(g(n))$ – някои леми

**Лема 16.** За всеки  $p > q > 0$  е в сила  $n^q = o(n^p)$ .

**Доказателство:**

Нека  $c > 0$ . Да потърсим  $n_0$  такова, че  $0 \leq n^q \leq c \cdot n^p, \forall n \geq n_0$ , т.е.  $1 \leq c \cdot n^{p-q}$  и значи  $n_0$  трябва да е  $\geq$  от  $(p-q)$ -ти корен от  $1/c$ .

**Лема 26.** За всяко  $a > 0$  е в сила  $\log_a n \in o(n)$ .

Доказателството на тази лема е по-трудно, заново ще се върнем на него по-късно.

# $O(g(n))$ – някои леми

**Лема 16.** За всеки  $p > q > 0$  е в сила  $n^q = o(n^p)$ .

**Доказателство:**

Нека  $c > 0$ . Да потърсим  $n_0$  такова, че  $0 \leq n^q \leq c \cdot n^p, \forall n \geq n_0$ , т.е.  $1 \leq c \cdot n^{p-q}$  и значи  $n_0$  трябва да е  $\geq$  от  $(p-q)$ -ти корен от  $1/c$ .

**Лема 26.** За всяко  $a > 0$  е в сила  $\log_a n \in o(n)$ .

Доказателството на тази лема е по-трудно, заново ще се върнем на него по-късно.

**Следствие.** За всяко  $a > 0$ ,  $n^p \cdot \log_a n \in o(n^{p+1})$  и в частност  $n \cdot \log_a n \in o(n^2)$ .

# $\Theta(g(n))$ - описание

**Дефиниция 3:** Казваме, че  $f(n)$  принадлежи на множеството  $\Theta(g(n))$ , ако  
**съществуват положителни константи  $c_1, c_2$  и  $n_0$**  такива, че

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0.$$

Вместо  $f(n) \in \Theta(g(n))$  може да пишем  $f(n) = \Theta(g(n))$ .

# $\Theta(g(n))$ - описание

**Дефиниция 3:** Казваме, че  $f(n)$  принадлежи на множеството  $\Theta(g(n))$ , ако  
**съществуват положителни константи  $c_1, c_2$  и  $n_0$**  такива, че

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0.$$

Вместо  $f(n) \in \Theta(g(n))$  може да пишем  $f(n) = \Theta(g(n))$ .

**Лема 1с.**  $\forall f(n) \in I_1$  е в сила  $f(n) = \Theta(f(n))$ .

# $\Theta(g(n))$ - описание

**Дефиниция 3:** Казваме, че  $f(n)$  принадлежи на множеството  $\Theta(g(n))$ , ако  
**съществуват положителни константи  $c_1, c_2$  и  $n_0$**  такива, че

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0.$$

Вместо  $f(n) \in \Theta(g(n))$  може да пишем  $f(n) = \Theta(g(n))$ .

**Лема 1с.**  $\forall f(n) \in I_1$  е в сила  $f(n) = \Theta(f(n))$ .

**Лема 2с.**  $f(n) \in \Theta(g(n))$  т.к.  $f(n) \in O(g(n))$  и  $g(n) \in O(f(n))$ .

# $\Theta(g(n))$ - описание

**Дефиниция 3:** Казваме, че  $f(n)$  принадлежи на множеството  $\Theta(g(n))$ , ако  
**съществуват положителни константи  $c_1, c_2$  и  $n_0$**  такива, че

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0.$$

Вместо  $f(n) \in \Theta(g(n))$  може да пишем  $f(n) = \Theta(g(n))$ .

**Лема 1с.**  $\forall f(n) \in I_1$  е в сила  $f(n) = \Theta(f(n))$ .

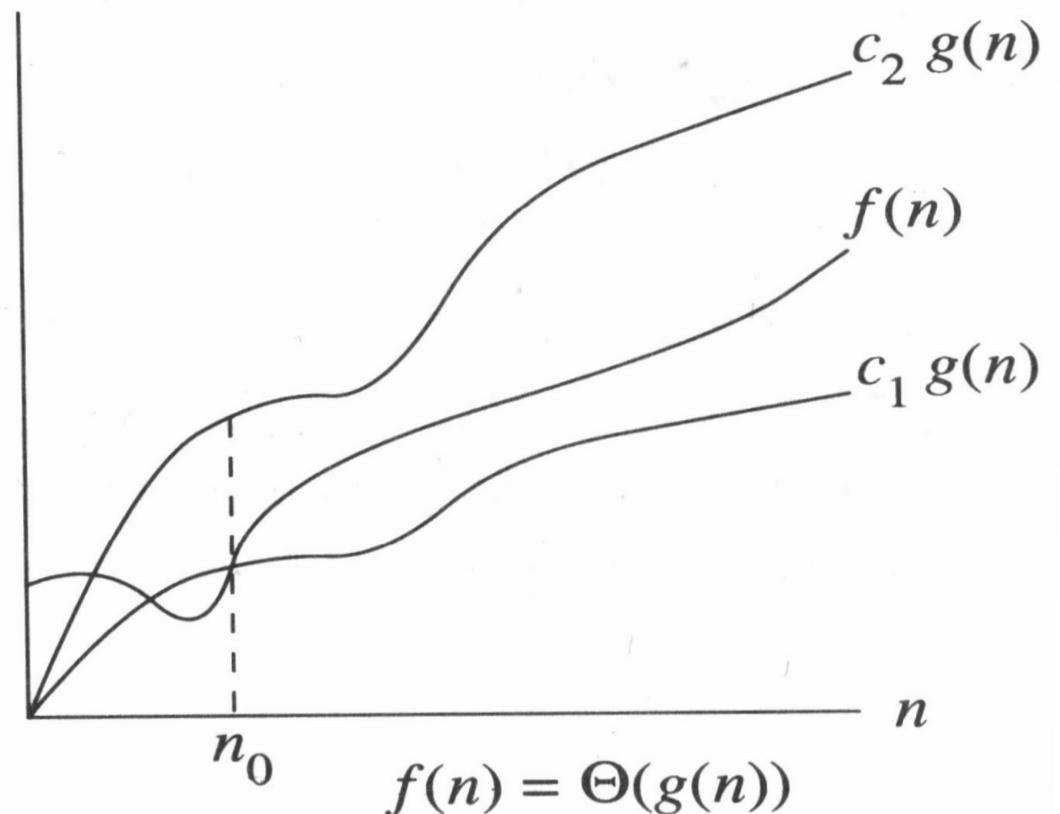
**Лема 2с.**  $f(n) \in \Theta(g(n))$  т.к.  $f(n) \in O(g(n))$  и  $g(n) \in O(f(n))$ .

**Пример:**  $2n^2 + 13 \in \Theta(n^2)$ . Защото вече показвахме, че  $2n^2 + 13 \in O(n^2)$  и  $n^2 \in O(2n^2 + 13)$ .

# $\Theta(g(n))$ - онагледяване

Виждаме как функцията  $f(n)$  от определен праг  $n_0$  нататък се установява между две мажорантите си. **Мажоранти наричаме функции, които „заграждат“ конкретна функция.**

**Чували ли сте за теорема за двамата полицаи?**



# $\Theta(g(n))$ – още леми

**Лема 3с.**  $f(n) \in \Theta(g(n))$  т.с.t к  $\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = const \neq 0$ .

## $\Theta(g(n))$ – още леми

**Лема 3с.**  $f(n) \in \Theta(g(n))$  т.к.  $\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = const \neq 0$ .

**Лема 4с.**  $f(n) \in o(g(n))$  т.к.  $\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = 0$ .

## $\Theta(g(n))$ – още леми

**Лема 3с.**  $f(n) \in \Theta(g(n))$  т.к.  $\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = const \neq 0$ .

**Лема 4с.**  $f(n) \in o(g(n))$  т.к.  $\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = 0$ .

**Лема 5с.**  $f(n) \in o(g(n))$  т.к.  $\lim_{x \rightarrow \infty} \frac{g(x)}{f(x)} = \infty$ .

# $\Theta(g(n))$ – още леми

**Лема 3с.**  $f(n) \in \Theta(g(n))$  т.к.  $\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = const \neq 0$ .

**Лема 4с.**  $f(n) \in o(g(n))$  т.к.  $\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = 0$ .

**Лема 5с.**  $f(n) \in o(g(n))$  т.к.  $\lim_{x \rightarrow \infty} \frac{g(x)}{f(x)} = \infty$ .

За да използваме горните леми ни трябва:

**Теорема (Лопитал):**

Ако  $\lim_{x \rightarrow \infty} f(x) = \infty$  и  $\lim_{x \rightarrow \infty} g(x) = \infty$ , или  $\lim_{x \rightarrow \infty} f(x) = 0$  и  $\lim_{x \rightarrow \infty} g(x) = 0$  тогава

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)}.$$

**Спомняте ли си за производните?**

# $\Theta(g(n))$ – важни примери

**Пример 1.**  $2n^2 + 13 \in \Theta(n^2)$ , защото  $\lim_{x \rightarrow \infty} \frac{2n^2 + 13}{n^2} = \lim_{x \rightarrow \infty} \frac{2 + \frac{13}{n^2}}{1} = 2$ .

# $\Theta(g(n))$ – важни примери

**Пример 1.**  $2n^2 + 13 \in \Theta(n^2)$ , защото  $\lim_{x \rightarrow \infty} \frac{2n^2 + 13}{n^2} = \lim_{x \rightarrow \infty} \frac{2 + \frac{13}{n^2}}{1} = 2$ .

**Пример 2.**  $2n^2 + 13 \in o(n^3)$ , защото  $\lim_{x \rightarrow \infty} \frac{2n^2 + 13}{n^3} = \lim_{x \rightarrow \infty} \frac{2 + \frac{13}{n^2}}{n} = 0$ .

# $\Theta(g(n))$ – важни примери

**Пример 1.**  $2n^2 + 13 \in \Theta(n^2)$ , защото  $\lim_{x \rightarrow \infty} \frac{2n^2 + 13}{n^2} = \lim_{x \rightarrow \infty} \frac{2 + \frac{13}{n^2}}{1} = 2$ .

**Пример 2.**  $2n^2 + 13 \in o(n^3)$ , защото  $\lim_{x \rightarrow \infty} \frac{2n^2 + 13}{n^3} = \lim_{x \rightarrow \infty} \frac{2 + \frac{13}{n^2}}{n} = 0$ .

**Лема.** Ако  $p < q$ , тогава  $n^p \in o(n^q)$  защото  $\lim_{x \rightarrow \infty} \frac{n^p}{n^q} = \lim_{x \rightarrow \infty} \frac{1}{n^{q-p}} = 0$ .

# $\Theta(g(n))$ – още примери

*Пример 3.*  $\ln n \in o(n)$ .

**Доказательство:**

# $\Theta(g(n))$ – още примери

**Пример 3.**  $\ln n \in o(n)$ .

**Доказательство:**

$$\lim_{x \rightarrow \infty} \frac{\ln n}{n} = \lim_{x \rightarrow \infty} \frac{(\ln n)'}{(n)'} = \lim_{x \rightarrow \infty} \frac{\frac{1}{n}}{1} = \lim_{x \rightarrow \infty} \frac{1}{n} = 0 .$$

# $\Theta(g(n))$ – още примери

**Пример 3.**  $\ln n \in o(n)$ .

**Доказательство:**

$$\lim_{x \rightarrow \infty} \frac{\ln n}{n} = \lim_{x \rightarrow \infty} \frac{(\ln n)'}{(n)'} = \lim_{x \rightarrow \infty} \frac{\frac{1}{n}}{1} = \lim_{x \rightarrow \infty} \frac{1}{n} = 0 .$$

**Пример 4.**  $\ln n \in o(\sqrt{n})$  ?

# $\Theta(g(n))$ – още примери

**Пример 3.**  $\ln n \in o(n)$ .

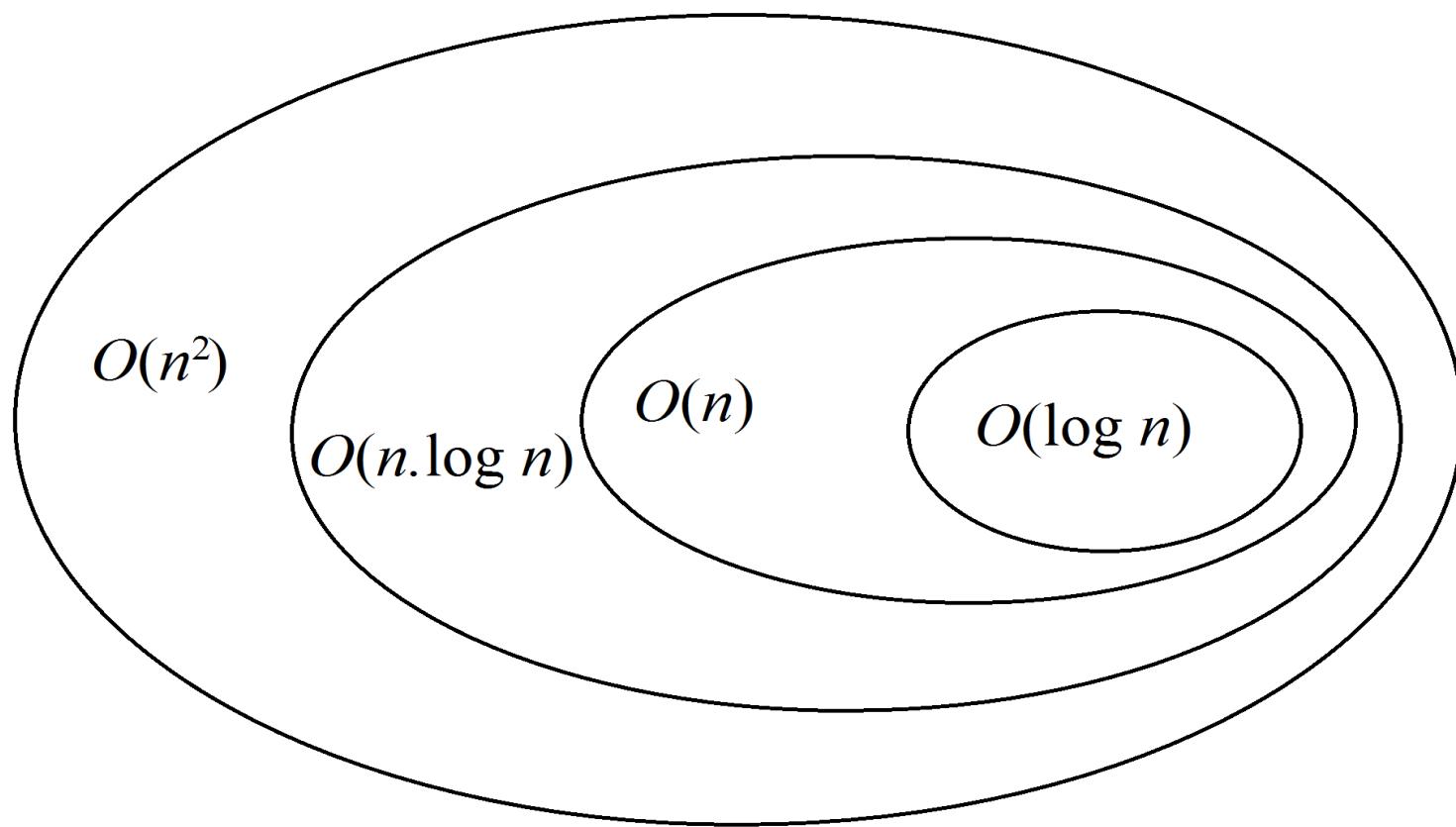
**Доказателство:**

$$\lim_{x \rightarrow \infty} \frac{\ln n}{n} = \lim_{x \rightarrow \infty} \frac{(\ln n)'}{(n)'} = \lim_{x \rightarrow \infty} \frac{\frac{1}{n}}{1} = \lim_{x \rightarrow \infty} \frac{1}{n} = 0 .$$

**Пример 4.**  $\ln n \in o(\sqrt{n})$  ?

Пресмятаме  $\lim_{x \rightarrow \infty} \frac{\ln n}{\sqrt{n}} = \lim_{x \rightarrow \infty} \frac{(\ln n)'}{(n^{\frac{1}{2}})'} = \lim_{x \rightarrow \infty} \frac{\frac{1}{n}}{\frac{1}{2}n^{-\frac{1}{2}}} = = \lim_{x \rightarrow \infty} \frac{2n^{\frac{1}{2}}}{n} = \lim_{x \rightarrow \infty} \frac{2}{n^{\frac{1}{2}}} = 0$ . Сл.  $\ln n \in o(\sqrt{n})$

# Йерархия на класовете



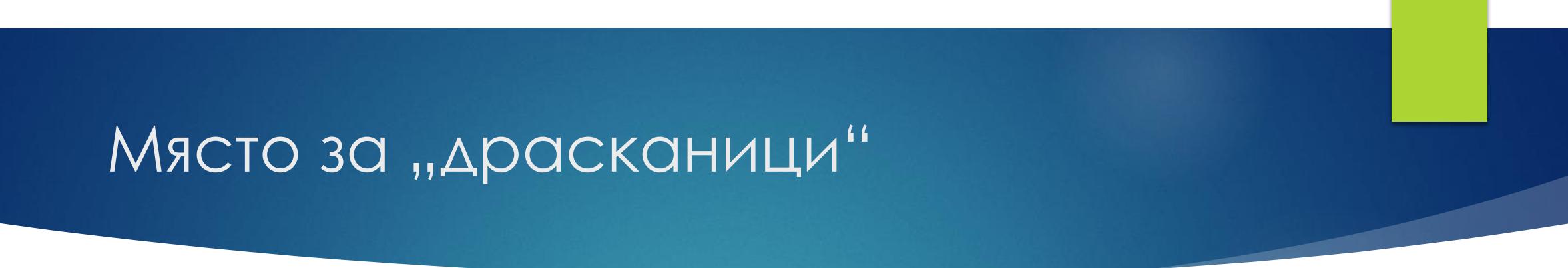
# Упражнения

- ▶ По зададени две функции да се провери някое от отношенията  $O()$ ,  $o()$  или  $\Theta()$  – например проверете за максималната сложност на метода на мехурчето.
- ▶ Например докажете, че  $\log_a x = \Theta(\log_b x)$ .
- ▶ Затова когато използваме логаритъм, за поведението на функцията няма значение при каква основа е.

# CSCB039 Алгоритми и програмиране

ЛЕКЦИЯ 5

гл. ас. д-р Слав Емилов Ангелов, НБУ



Място за „драсканици“

# Любопитно – Merge sort без памет

Оказва се, че през 2008 година е измислена техника, която успява да реализира **Merge sort с два пъти по-малко допълнителна памет**, а в последствие и **без допълнителна памет**. Още по-удивителното е, че **изчислителната сложност по време се запазва в  $O(n\log n)$ !**

Линк: <https://github.com/BonzaiThePenguin/WikiSort>

От линка ще намерите информация и код за **WikiSort**, **Greilsort** и **Block merge sort**. Всички тези алгоритми имат сходства и разлики, но Block merge sort е ядрото. Допълнително се коментира **Bottom-up merge sort**, който премахва рекурсивните извиквания. Коментира се и как да комбинираме наредените подмасиви, че да са сходни по размер (+10% ефикасност).

# Квадратни числа

Свойства:

- ▶  $s(n) = n^2 = (n - 1)^2 + (2n - 1) = \sum_{i=1}^n (2i - 1)$ ;
- ▶ Всяко нечетно число на квадрат е нечетно число, понеже  $(2n + 1)^2 = 4(n^2 + n) + 1$ ;
- ▶ Ясно е, че и всяко четно число на квадрат е четно число;
- ▶ Числа във формата  **$4n+2, 4n+3$**  не могат да са точни квадрати;
- ▶ Всяко **квадратно число** е сума на две последователни **триъгълни числа**;
- ▶ Сума на две последователни **квадратни числа** е **центрирано квадратно число**.

$$m = 1^2 = 1$$



$$m = 2^2 = 4$$



$$m = 3^2 = 9$$



$$m = 4^2 = 16$$



$$m = 5^2 = 25$$



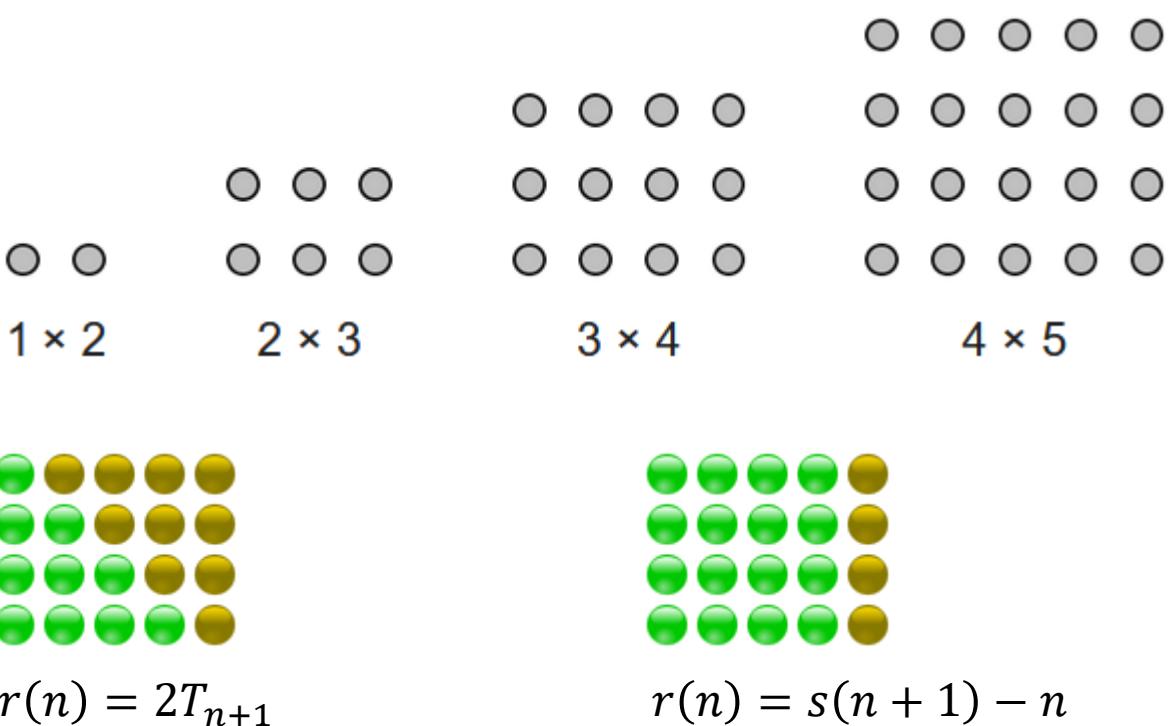
**Чували ли сте за „безквадратни числа“?**

# Правоъгълни числа (Pronic number)

Свойства:

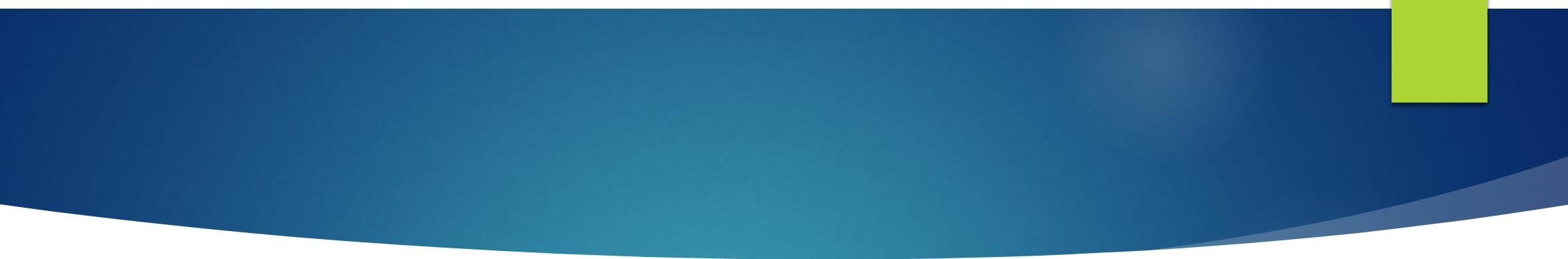
- ▶  $r(n) = n(n + 1);$
- ▶  $\sum_{i=1}^n r(i) = \sum_i^n i(i + 1) = \frac{n(n+1)(n+2)}{3};$
- ▶  $\sum_{i=1}^n \frac{1}{r(i)} = \frac{n}{n+1} \xrightarrow{n \rightarrow \infty} 1;$
- ▶  $\frac{r(n)+r(n+1)}{2} = s(n + 1);$
- ▶ Ако допълним някое  $r(n)$  с 25, резултата ще е квадратно число, защото:

$$100n(n + 1) + 25 = (10n + 5)^2.$$



# Задача

Оценете към кои класове на сложност се причислява класическия Merge sort по отношение на брой сравнения при  $2^k$  елемента (това са оптималните случаи за подразделянето).



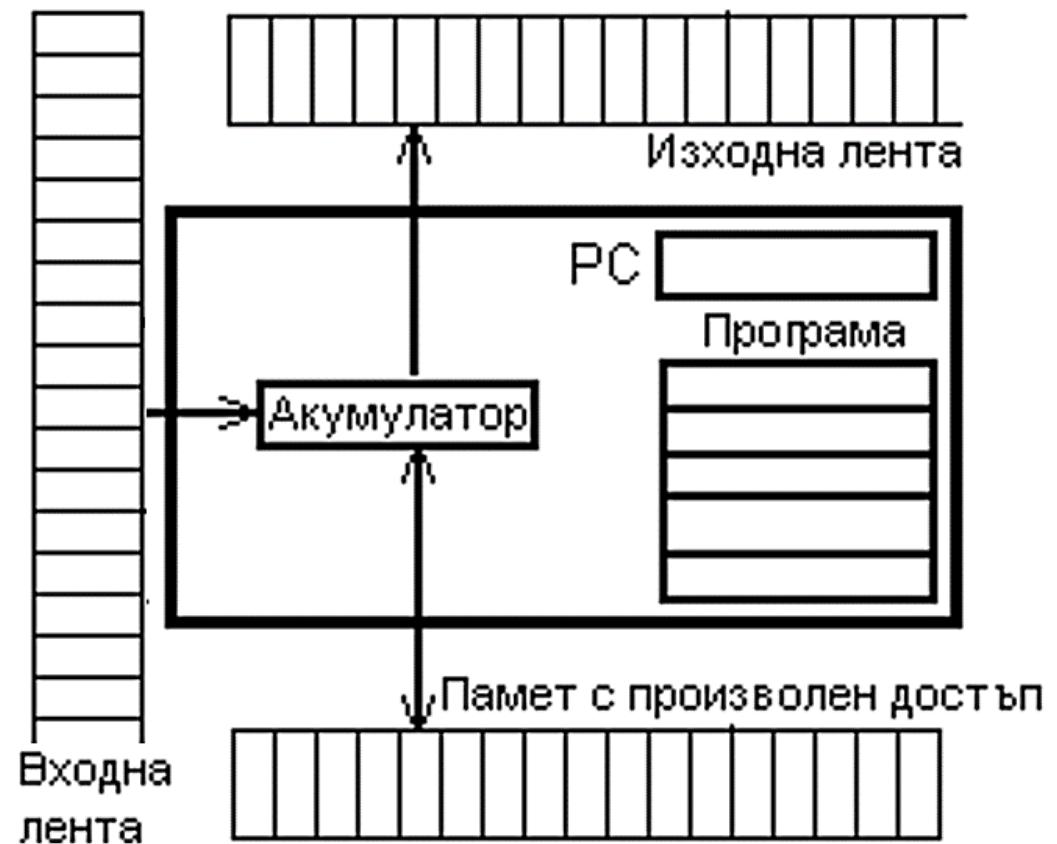
# Формализми за понятието алгоритъм

# Нуждата от формални модели

- ▶ Понятието алгоритъм не е формално математическо понятие, а едно от изискванията, за да бъде една процедура алгоритмична е да е формална;
- ▶ За да можем да изследваме алгоритмите е необходимо да ги представяме със средствата на някаква формална, математически дефинирана изчислителна среда – формализъм;
- ▶ Известни са много различни формализми за представяне на алгоритми – машини на Тюринг, машини на Пост, рекурсивни функции,  $\lambda$ -смятане, машини с произволен достъп до паметта и др.

# Машина с произволен достъп до паметта

Машината с произволен достъп до паметта (МПД) е опростен модел на съвременните компютри:



# Машина с произволен достъп до паметта – управляващ блок

**Управляващият блок** може да изпълнява множество **команди**. Той се състои от:

- ▶ Два **регистъра**, всеки от които може да съдържа цяло число от някакъв тип, да го наречем **num**:
  - 1) **Акумулатор AC** – с него се извършват операциите;
  - 2) **Брояч на команди PC** - съдържа адреса на изпълняваната команда.
- ▶ **Памет за програмите** - крайно множество от **клетки**, номерирани с цели положителни числа от беззнаков тип. Да го означим с **adr**,  $adr = \text{unsigned num}$  – **адрес на командата**. Всяка клетка може да съдържа по една команда .

# Машина с произволен достъп до паметта – входна лента

**Входната лента** е безкрайна в едната посока редица от клетки от тип num. Там се намират входните данни.

- ▶ Тази лентата е снабдена с **четяща глава**, която винаги се намира над точно една клетка на лентата и може да прочете съдържанието ѝ в АС, при изпълнение на съответна команда;
- ▶ След прочитането главата **се измества на една клетка** в посока безкрайността;
- ▶ В началото главата е **върху първата клетка** на входната лента.

# Машина с произволен достъп до паметта - изходна лента

**Изходната лента** е безкрайна в едната посока редица от клетки от тип пум.

- ▶ Тази лентата е снабдена с **пишеща глава**, която винаги се намира над точно една клетка на лентата и може да запише в нея съдържанието на АС, при изпълнение на съответна команда;
- ▶ След прочитането главата **се измества на една клетка** в посока безкрайността;
- ▶ В началото главата е **върху първата клетка** на входната лента .

# Машина с произволен достъп до паметта – памет с произволен достъп

**Паметта с произволен достъп (ПД)** е крайно множество от **клетки**, номерирани с цели положителни числа от adr. Всяко такова число наричаме **адрес в паметта с ПД**. Всяка клетка може да съдържа по едно число от тип num.

- ▶ Тази лентата е снабдена с **четящо-пишеща глава** и при изпълнение на съответна команда се премества без загуба на време над зададена клетка на лентата и може да прочете съдържанието ѝ в АС или да запише в нея съдържанието на АС.

# Машина с произволен достъп до паметта - програма

- ▶ Всяка команда от програмата на МПД има свой **код** (вместо обичайните числа използваме мнемонични низове за код) и може да има или да няма **аргумент**.
- ▶ **Командите** се разполагат в програмната памет, започвайки **от адрес 0** и без да се оставят празни клетки. Всяка команда на програмата получава свой адрес, който при писане на програмата записваме преди нея, последван от дясна скоба. Така общий вид на една команда е  
адрес) код [ аргумент ],

където със знаците [...] означаваме, че някои от командите не се нуждаят от аргумент.

# Машина с произволен достъп до паметта – команди с пряка адресация

- ▶ **Команди с пряка адресация.** При тези команди, **операндът** на предизвиканата от командата операция се намира **в памета за данни**, а в командата като аргумент се поставя адресът му.
- ▶ Командите с пряка адресация задаваме без специален знак в края на мнемониката. Например, при изпълнение на

1000) ADD 245

съдържанието на АС ще се събере с числото, съдържащо се в клетката с адрес 245 и ще замени в АС старото съдържание. Това действие означаваме така

$\langle AC \rangle := \langle AC \rangle + \langle A \rangle,$

където А е адресът на операнда.

- ▶ След изпълнението на такава команда РС се увеличава с 1 и се преминава към изпълнение на командата разположена в следващата клетка, т.е.  $\langle PC \rangle := \langle PC \rangle + 1$ .

# Машина с произволен достъп до паметта – команди с непосредствен operand

- ▶ **Команди с непосредствен operand.** При тези команди, operandът на предизвиканата от командата операция се задава като аргумент.
- ▶ Командите с пряка адресация задаваме със знак # в края на мнемониката. Например, при изпълнение на

1000) ADD# 245

съдържанието на АС ще се събере с числото 245 и ще замени в АС старото съдържание. Това действие означаваме с

$\langle AC \rangle := \langle AC \rangle + A,$

където A е аргументът.

- ▶ След изпълнението на такава команда РС също се увеличава с 1, т.е.  $\langle PC \rangle := \langle PC \rangle + 1.$

# Машина с произволен достъп до паметта – команди с косвена адресация

- ▶ **Команди с косвена адресация.** При тези команди, операндът на предизвиканата от командата операция се намира **в памета за данни**, а в командата като аргумент се поставя **адрес на клетка, която съдържа адреса на операнда**.
- ▶ Командите с пряка адресация задаваме със знак @ в края на мнемониката. Например, при изпълнение на

1000) ADD@ 245

съдържанието на АС ще се събере с числото, съдържащо се в клетката, чийто адрес се намира в клетка с адрес 245 и ще замени в АС старото съдържание. Това действие означаваме така

$\langle AC \rangle := \langle AC \rangle + \langle A \rangle,$

където А е адресът на operand-a.

- ▶ И в този случай  $\langle PC \rangle := \langle PC \rangle + 1$ .

# Машина с произволен достъп до паметта – команди (1/2)

| Команда  | Действие   |
|----------|--|
| LOAD# I  | $\langle AC \rangle := I; \langle PC \rangle := \langle PC \rangle + 1$  |
| LOAD A   | $\langle AC \rangle := \langle A \rangle; \langle PC \rangle := \langle PC \rangle + 1$                                      |
| LOAD@ A  | $\langle AC \rangle := \langle \langle A \rangle \rangle; \langle PC \rangle := \langle PC \rangle + 1$                      |
| STORE A  | $\langle A \rangle := \langle AC \rangle; \langle PC \rangle := \langle PC \rangle + 1$                                      |
| STORE@ A | $\langle \langle A \rangle \rangle := \langle AC \rangle; \langle PC \rangle := \langle PC \rangle + 1$                      |
| ADD# I   | $\langle AC \rangle := \langle AC \rangle + I; \langle PC \rangle := \langle PC \rangle + 1$                                 |
| ADD A    | $\langle AC \rangle := \langle AC \rangle + \langle A \rangle; \langle PC \rangle := \langle PC \rangle + 1$                 |
| ADD@ A   | $\langle AC \rangle := \langle AC \rangle + \langle \langle A \rangle \rangle; \langle PC \rangle := \langle PC \rangle + 1$ |
| SUB# I   | $\langle AC \rangle := \langle AC \rangle - I; \langle PC \rangle := \langle PC \rangle + 1$                                 |
| SUB A    | $\langle AC \rangle := \langle AC \rangle - \langle A \rangle; \langle PC \rangle := \langle PC \rangle + 1$                 |
| SUB@ A   | $\langle AC \rangle := \langle AC \rangle - \langle \langle A \rangle \rangle; \langle PC \rangle := \langle PC \rangle + 1$ |
| MUL# I   | $\langle AC \rangle := \langle AC \rangle * I; \langle PC \rangle := \langle PC \rangle + 1$                                 |
| MUL A    | $\langle AC \rangle := \langle AC \rangle * \langle A \rangle; \langle PC \rangle := \langle PC \rangle + 1$                 |
| MUL@ A   | $\langle AC \rangle := \langle AC \rangle * \langle \langle A \rangle \rangle; \langle PC \rangle := \langle PC \rangle + 1$ |

# Машина с произволен достъп до паметта – команди (2/2)

|        |   |
|--------|---|
| DIV# I | $\langle AC \rangle := \langle AC \rangle / I; \langle PC \rangle := \langle PC \rangle + 1$                                  |
| DIV A  | $\langle AC \rangle := \langle AC \rangle / \langle A \rangle; \langle PC \rangle := \langle PC \rangle + 1$                  |
| DIV@ A | $\langle AC \rangle := \langle AC \rangle / \langle \langle A \rangle \rangle; \langle PC \rangle := \langle PC \rangle + 1$  |
| MOD# I | $\langle AC \rangle := \langle AC \rangle \% I; \langle PC \rangle := \langle PC \rangle + 1$                                 |
| MOD A  | $\langle AC \rangle := \langle AC \rangle \% \langle A \rangle; \langle PC \rangle := \langle PC \rangle + 1$                 |
| MOD@ A | $\langle AC \rangle := \langle AC \rangle \% \langle \langle A \rangle \rangle; \langle PC \rangle := \langle PC \rangle + 1$ |
| JMP B  | $\langle PC \rangle := B$   |
| JMPZ B | Ако $\langle AC \rangle = 0$ , то $\langle PC \rangle := B$ , иначе $\langle PC \rangle := \langle PC \rangle + 1$            |
| JMPP B | Ако $\langle AC \rangle > 0$ , то $\langle PC \rangle := B$ , иначе $\langle PC \rangle := \langle PC \rangle + 1$            |
| JMPN B | Ако $\langle AC \rangle < 0$ , то $\langle PC \rangle := B$ , иначе $\langle PC \rangle := \langle PC \rangle + 1$            |
| INPUT  | Поредната клетка на входната лента се прочита в $AC$ , главата се мести на следваща клетка                                    |
| OUTPUT | $\langle AC \rangle$ се записва в поредна клетка на изходната лента, а главата се мести на следваща клетка                    |
| STOP   | Прекратява изпълнението на програмата   |

# Машина с произволен достъп до паметта - пример

**Пример.** Нека напишем програма за МПД, която въвежда от входната лента коефициентите  $a$ ,  $b$  и  $c$  на квадратно уравнение, пресмята дискриминантата  $D$  на уравнението по формулата  $b^2-4ac$  и извежда получения резултат на изходната лента.

**Решение.** Първата работа при решаването на задача с МПД е да определим къде ще поставим данните:

- 0) за коефициента  $a$ ,
- 1) за коефициента  $b$ ,
- 2) за коефициента  $c$ ,
- 3) за дискриминантата  $D$ ,
- 4) за междинен резултат.

# Машина с произволен достъп до паметта – решение на примера

Програмата, коята решава задачата:

```
0) INPUT      // въвеждаме а в AC
1) STORE 0    // съхраняваме а в паметта
2) INPUT      // въвеждаме б в AC
3) STORE 1    // съхраняваме б в паметта
4) INPUT      // въвеждаме с в AC
5) STORE 2    // съхраняваме с в паметта
6) LOAD# 4    // поставяме 4 в AC
7) MUL 0      // умножаваме AC по а
8) MUL 2      // умножаваме AC по с
9) STORE 4    // запазваме 4ас в паметта
10) LOAD 1     // поставяме б в AC
11) MUL 1      // умножаваме AC по б
12) SUB 4      // изваждаме 4ас от AC
13) STORE 3    // запазваме резултата
14) OUTPUT     // извеждаме резултата
15) STOP       // прекратяваме изпълнението
```

# Задача

Какво ще направи  
програмата, коята  
е показана вдясно?

- 0) LOAD# 1
- 1) STORE 0
- 2) INPUT
- 3) JMPZ 9
- 4) STORE@ 0
- 5) LOAD 0
- 6) ADD# 1
- 7) STORE 0
- 8) JMP 2
- 9) LOAD 0
- 10) SUB# 1
- 11) STORE 0
- 12) OUTPUT
- 13) STOP

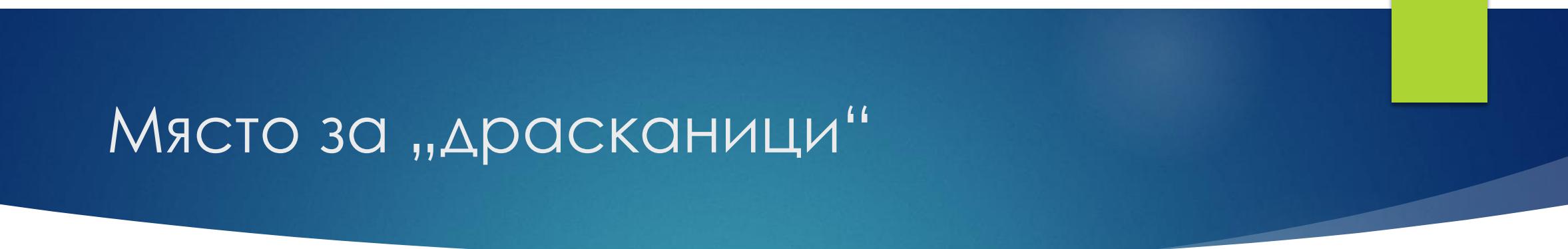
## Задача 2

Напишете програма за МПД, която въвежда от лентата дължините на двете страни на правоъгълник, пресмята и извежда на изходната лента периметъра и лицето на правоъгълника.

# CSCB039 Алгоритми и програмиране

ЛЕКЦИЯ 6

гл. ас. д-р Слав Емилов Ангелов, НБУ



Място за „драсканици“

# Задача - решение

Оценете към кои класове на сложност се причислява класическия Merge sort по отношение на брой сравнения при  $n = 2^k$  елемента (това са оптималните случаи за подразделянето).

| Всяко квадратче е вече нареден блок от елементи |       |                         |                                |  |  |
|---|-------|-------------------------|--------------------------------|--|--|
| к)  |       | $2^{k-k}$ слиивания     | макс $2^{k-0}-1$ сравнения     |  |  |
| к-1)  |       | $2^{k-(k-1)}$ слиивания | макс $2^{k-1}-1$ сравнения     |  |  |
| к-2)  |       | $2^{k-(k-2)}$ слиивания | макс $2^{k-2}-1$ сравнения     |  |  |
| к-3)  |       | $2^{k-(k-3)}$ слиивания | макс $2^{k-3}-1$ сравнения     |  |  |
| ...   | ..... | ...                     | ...                            |  |  |
| 1)  |       | $2^{k-1}$ слиивания     | макс $2^{k-(k-1)}-1$ сравнения |  |  |

# Решение

Остава да умножим броя на сливания на всяко ниво по броя сравнения за сливане в конкретното ниво:

$$\begin{aligned} \sum_{i=1}^k 2^{k-i} (2^{k-(k-i)} - 1) &= \sum_{i=1}^k (2^k - 2^{k-i}) = \sum_{i=1}^k 2^k - \sum_{i=1}^k 2^{k-i} = \sum_{i=1}^k 2^k - 2^k \sum_{i=1}^k \frac{1}{2^i} = k \cdot 2^k - 2^k \sum_{i=1}^k \frac{1}{2^i} = \\ &= k \cdot 2^k - 2^k \frac{\frac{1}{2} \left(1 - \left(\frac{1}{2}\right)^k\right)}{1 - \frac{1}{2}} = k \cdot 2^k - 2^k \left(1 - \left(\frac{1}{2}\right)^k\right) = k \cdot 2^k - 2^k + 1 = n \log_2 n - n + 1 \rightarrow O(n \log n). \end{aligned}$$

$n = 2^k \rightarrow k = \log_2 n.$

# Език за програмиране

- ▶ Всеки **Език за процедурно програмиране** е формален механизъм за представяне на алгоритмични процедури;
- ▶ Програмите, написани на езици за логическо и функционално програмиране представлят по-скоро **математическата същност на задачата**, а не алгоритъм за нейното решаване;
- ▶ Ще използваме **езика С**, но всички разсъждения ще бъдат аналогични при всеки друг език за процедурно програмиране.

# Език за програмиране

Ще покажем как всяка от конструкциите на езика С може да се преведе в програмен фрагмент на МПД така, че сложността на една програма написана на С **да е равна на сложността на съответната и програма, написана за МПД**, с точност до някакви константи.

# Машина с произволен достъп до паметта – команди (1/2)

| Команда  | Действие   |
|----------|--|
| LOAD# I  | $\langle AC \rangle := I; \langle PC \rangle := \langle PC \rangle + 1$  |
| LOAD A   | $\langle AC \rangle := \langle A \rangle; \langle PC \rangle := \langle PC \rangle + 1$                                      |
| LOAD@ A  | $\langle AC \rangle := \langle \langle A \rangle \rangle; \langle PC \rangle := \langle PC \rangle + 1$                      |
| STORE A  | $\langle A \rangle := \langle AC \rangle; \langle PC \rangle := \langle PC \rangle + 1$                                      |
| STORE@ A | $\langle \langle A \rangle \rangle := \langle AC \rangle; \langle PC \rangle := \langle PC \rangle + 1$                      |
| ADD# I   | $\langle AC \rangle := \langle AC \rangle + I; \langle PC \rangle := \langle PC \rangle + 1$                                 |
| ADD A    | $\langle AC \rangle := \langle AC \rangle + \langle A \rangle; \langle PC \rangle := \langle PC \rangle + 1$                 |
| ADD@ A   | $\langle AC \rangle := \langle AC \rangle + \langle \langle A \rangle \rangle; \langle PC \rangle := \langle PC \rangle + 1$ |
| SUB# I   | $\langle AC \rangle := \langle AC \rangle - I; \langle PC \rangle := \langle PC \rangle + 1$                                 |
| SUB A    | $\langle AC \rangle := \langle AC \rangle - \langle A \rangle; \langle PC \rangle := \langle PC \rangle + 1$                 |
| SUB@ A   | $\langle AC \rangle := \langle AC \rangle - \langle \langle A \rangle \rangle; \langle PC \rangle := \langle PC \rangle + 1$ |
| MUL# I   | $\langle AC \rangle := \langle AC \rangle * I; \langle PC \rangle := \langle PC \rangle + 1$                                 |
| MUL A    | $\langle AC \rangle := \langle AC \rangle * \langle A \rangle; \langle PC \rangle := \langle PC \rangle + 1$                 |
| MUL@ A   | $\langle AC \rangle := \langle AC \rangle * \langle \langle A \rangle \rangle; \langle PC \rangle := \langle PC \rangle + 1$ |

# Машина с произволен достъп до паметта – команди (2/2)

|        |   |
|--------|---|
| DIV# I | $\langle AC \rangle := \langle AC \rangle / I; \langle PC \rangle := \langle PC \rangle + 1$                                  |
| DIV A  | $\langle AC \rangle := \langle AC \rangle / \langle A \rangle; \langle PC \rangle := \langle PC \rangle + 1$                  |
| DIV@ A | $\langle AC \rangle := \langle AC \rangle / \langle \langle A \rangle \rangle; \langle PC \rangle := \langle PC \rangle + 1$  |
| MOD# I | $\langle AC \rangle := \langle AC \rangle \% I; \langle PC \rangle := \langle PC \rangle + 1$                                 |
| MOD A  | $\langle AC \rangle := \langle AC \rangle \% \langle A \rangle; \langle PC \rangle := \langle PC \rangle + 1$                 |
| MOD@ A | $\langle AC \rangle := \langle AC \rangle \% \langle \langle A \rangle \rangle; \langle PC \rangle := \langle PC \rangle + 1$ |
| JMP B  | $\langle PC \rangle := B$   |
| JMPZ B | Ако $\langle AC \rangle = 0$ , то $\langle PC \rangle := B$ , иначе $\langle PC \rangle := \langle PC \rangle + 1$            |
| JMPP B | Ако $\langle AC \rangle > 0$ , то $\langle PC \rangle := B$ , иначе $\langle PC \rangle := \langle PC \rangle + 1$            |
| JMPN B | Ако $\langle AC \rangle < 0$ , то $\langle PC \rangle := B$ , иначе $\langle PC \rangle := \langle PC \rangle + 1$            |
| INPUT  | Поредната клетка на входната лента се прочита в $AC$ , главата се мести на следваща клетка                                    |
| OUTPUT | $\langle AC \rangle$ се записва в поредна клетка на изходната лента, а главата се мести на следваща клетка                    |
| STOP   | Прекратява изпълнението на програмата   |

# Сложност на израз

Правило:

**Сложността**  $t_{\{expr\}}$  на израза  $expr$  е равна на  $1 +$  сумата от сложностите на участващите в израза индексни изрази, като броя на знаците за операции, броя на задължителните кръгли скоби (ако не ползваме приоритет) и броя на индексните скоби.

| Пример 1                            | Пример 2   | Пример 3  | Пример 4  |
|-------------------------------------|--|---|---|
| $a + b - c$                         | $a + b * c =$<br>$a + (b * c)$                                   | $a * b + (c * d)$   | $a[b] + c * d =$<br>$a[b] + (c * d)$  |
| a) LOAD a<br>+1) ADD b<br>+2) SUB c | a) LOAD b<br>+1) MUL c<br>+2) STORE x<br>+3) LOAD a<br>+4) ADD x | a) LOAD c<br>+1) MUL d<br>+2) STORE x<br>+3) LOAD a<br>+4) MUL b<br>+5) ADD x | a) LOAD c<br>+1) MUL d<br>+2) STORE x<br>+3) LOAD a<br>+4) ADD b<br>+5) STORE y<br>+6) LOAD@ y<br>+7) ADD x |

# Сложност на израз

Правило:

**Сложността**  $t_{\{assgn\}}$  на израза  $assgn$  е равна на сложността на съответния израз.

Правило:

**Сложността на израз, в който има операция за сравняване**

трябва да се увеличи с 1, заради неизбежната команда за преход.

Например:

$a > b$  се превежда в:

- a) LOAD a
- +1) SUB b
- +2) JMP<sub>P</sub> x

| Пример 1                                     | Пример 2  |
|--|---|
| $a[b] = c * d;$                              | $a++; \quad a--; \quad a += b; \quad a -= b; \dots$<br>$(a = a + 1; \quad a = a - 1; \quad a = a + b; \dots)$ |
| a)<br>+1)<br>+2)<br>LOAD a<br>ADD b<br>SUB c | a)<br>+1)<br>+2)<br>LOAD a<br>ADD# 1<br>STORE a   |

Коригирайте операциите

# Сложност на израз

Правило:

**Сложността**  $t_{\{assgn\}}$  на израза  $assgn$  е равна на сложността на съответния израз.

Правило:

**Сложността на израз, в който има операция за сравняване**

трябва да се увеличи с 1, заради неизбежната команда за переход.

Например:

$a > b$  се превежда в:

- a) LOAD a
- +1) SUB b
- +2) JMP<sub>P</sub> x

| Пример 1                                     | Пример 2  |
|--|---|
| $a[b] = c * d;$                              | $a++; \quad a--; \quad a += b; \quad a -= b; \dots$<br>$(a = a + 1; \quad a = a - 1; \quad a = a + b; \dots)$ |
| a)<br>+1)<br>+2)<br>LOAD a<br>ADD b<br>SUB c | a)<br>+1)<br>+2)<br>LOAD a<br>ADD# 1<br>STORE a<br>a)<br>+1)<br>+2)<br>LOAD a<br>ADD b<br>STORE a             |

Коригирайте операциите:

- a) LOAD a
- +1) ADD b
- +2) STORE tmp
- +3) LOAD c
- +4) MUL d
- +5) STORE@ tmp

# СЛОЖНОСТ НА ИЗРАЗ С &&

| Пример 1  | Пример 2   |
|---|--|
| $a>b \&\& c>d$  | $a[i]>b \&\& d[j]<c$   |
| Приключваме,<br>ако първото<br>условие не е<br>изпълнено                    | 0) LOAD a<br>+1) SUB b<br>+2) JMPNZ x1<br>+3) LOAD c<br>+4) SUB d<br>+5) JMPP x2 |
| Ако и второто е<br>изпълнено,<br>влизаме в<br>съответния блок<br>от команди | Общо: 6  |

# БЛОК ОТ ОПЕРАТОРИ

Правило:

**Сложността  $t_{\{block\}}$  на блока от оператори:**

```
{  
    oper_1;  
    oper_2;  
    . . .  
    oper_k;  
}
```

$$\text{e } t_{\{block\}} = t_{\{oper\_1\}} + t_{\{oper\_2\}} + \dots + t_{\{oper\_k\}}.$$

# Условен оператор

| <b>if</b>  | <b>If else</b>  |
|--|---|
| <pre>if (израз) then<br/>    оператор</pre>                                | <pre>if (израз) then<br/>    оператор1<br/>else<br/>    оператор2</pre>   |
| <pre>t_{if_then}(n) =<br/>    t_{израз}(n) +<br/>    t_{оператор}(n)</pre> | <pre>t_{if_then}(n) =<br/>    t_{израз}(n) +<br/>    max[t_{оператор1}(n),<br/>         t_{оператор2}(n)]</pre> |

# Оператори while и do...while

При пресмятане на сложността на цикъл може да го направим грубо, като оценим колко пъти ще се изпълни оператора. Нека означим броя итерации с  $c(n)$ .

| while (израз)<br>оператор  | do<br>оператор<br>while (израз);   |
|--|--|
| $t_{\{\text{while}\}}(n) = [c(n)+1] * t_{\{\text{израз}\}}(n) + c(n) * t_{\{\text{оператор}\}}(n)$ | $t_{\{\text{do\_while}\}}(n) = c(n) * t_{\{\text{израз}\}}(n) + c(n) * t_{\{\text{оператор}\}}(n)$ |

# Оператори while и do...while

Пресмятането на сложността на цикъл може да се направим и прецизно, като оценим сложността на всяко от  $c(n)$ -те итерации

while (израз)  
оператор

$$\begin{aligned}t_{\text{while}}(n) &= \\&= [c(n)+1] * t_{\text{израз}}(n) + \\&+ t_{\text{опер}_{[1]}}(n) + t_{\text{опер}_{[2]}}(n) + \\&+ \dots + t_{\text{опер}_{[c(n)]}}(n)\end{aligned}$$

do  
оператор  
while (израз);

$$\begin{aligned}t_{\text{do\_while}}(n) &= \\&= c(n) * t_{\text{израз}}(n) + \\&+ t_{\text{опер}_{[1]}}(n) + t_{\text{опер}_{[2]}}(n) + \\&+ \dots + t_{\text{опер}_{[c(n)]}}(n)\end{aligned}$$

# Оператор for – груба оценка

И в този случай можем да подходим грубо

```
for (израз1; израз2; израз3)  
    оператор
```

$$\begin{aligned}t_{\text{for}}(n) = & t_{\text{израз1}}(n) + \\& + [c(n)+1] * t_{\text{израз2}}(n) + \\& + c(n) * t_{\text{израз3}}(n) + \\& + c(n) * t_{\text{оператор}}(n)\end{aligned}$$

# Оператор for – прецизна оценка

Или да подходим прецизно

for (израз1; израз2; израз3)

оператор

```
t_{for}(n) = t_{израз1}(n) +
+ [c(n)+1] * t_{израз2}(n) +
+ c(n) * t_{израз3}(n) +
+ t_{опер[1]}(n) + t_{опер[2]}(n) +
...
+ t_{опер[c(n)]}(n)
```

# Оператор for - пример

## Пример 1

```
min = a[0]; t(n) = 5
      2   3   3
for (i=1; i<n; i++)
      6           5
    if(a[i]<min) min = a[i];
```

```
t_{for}(n) = 2 +
  + n * 3 +
  + (n-1) * 3 +
  + (n-1) * (6+5) = 17n - 11
```

```
t_{for}(n) = c1 + n * c2 +
  + (n-1) * c3 + (n-1) * c4
= D1*n + D2
```

# Оператор for - Пример 2

```
int a[], n;  
void bubble_sort()  
{  
    int i,j,t;  
    for(i=n-13; i>=03; i--3)  
        for(j=02; j<i3; j++3)  
            if(a[j]>a[j+1]10)  
            { t=a[j]5; a[j]=a[j+1]9; a[j+1]=t6; }  
}
```

```
LOAD a  
ADD j  
STORE x1  
LOAD a  
ADD j  
ADD# 1  
STORE x2  
LOAD@ x1  
SUB@ x2  
JMPP x0  
  
LOAD a  
ADD j  
ADD# 1  
STORE x  
LOAD t  
STORE@ x  
  
LOAD a  
ADD j  
STORE x1  
LOAD a  
ADD j  
ADD# 1  
STORE x2  
LOAD@ x2  
STORE@ x1
```

# Оператор for – Пример 2

```
int a[], n;  
  
void bubble_sort()  
{ int i,j,t;  
    for(i=n-13; i>=03; i--3)      3+(3n+3)+3n  
        for(j=02; j<i3; j++3)      2+(3i+3)+3i  
            if(a[j]>a[j+1]10)          10i  
            { t=a[j]5; a[j]=a[j+1]9;      30i } }  
                a[j+1]=t6; }      20i  
    }  
                                за i=n-1,n-1,...,0
```

# Оператор for – крайна оценка

$$\begin{aligned}t_{\text{bubble\_sort}}(n) &= \\&= 6n + 6 + \sum_{i=0,1,\dots,n-1} (36i+5) = \\&= 6n + 6 + 36 \cdot \sum_{i=0,1,\dots,n-1} i + 5 \cdot \sum_{i=0,1,\dots,n-1} 1 = \\&= 6n + 6 + 36n(n-1)/2 + 5n = 6n + 6 + 18n^2 + 18n + 5n = \\&= \mathbf{18n^2 + 29n + 6} = \mathbf{O(n^2)}.\end{aligned}$$

$\sum_{i=0,1,\dots,n-1} i = 1+2+\dots+(n-1) = n(n-1)/2.$

# Извикване на нерекурсивна функция

- ▶ При извикване на нерекурсивна функция оценяваме сложността ѝ както на главната функция;
- ▶ Разликата е, че за целта трябва да намерим **размера  $n'(n)$**  на „входните“ за тази функция данни като функция от размера  $n$  на входните данни на програмата;
- ▶ В примера по-горе, ако оформим намирането на минимум на масив, с който работим като функция, тогава  $n' = n$ .

# Сложност на С програма

**Сложност на С програма е сложността на главната ѝ функция.**

# Интересен пример - ЕВКЛИД

Да се намери НОД на  $a \geq b > 0$ .

```
int gcd(int a, int b)
{
    int x,y,r;
    x=a; y=b;
    do {
        r=x%y; x=y; y=r;
    } while(r!=0);

    return x;
}
```

# Интересен пример - Евклид

| x  | y  |
|----|----|
| 21 | 13 |
| 13 | 8  |
| 8  | 5  |
| 5  | 3  |
| 3  | 2  |
| 2  | 1  |
| 1  | 1  |
| 1  | 0  |

- ▶ Теорема: На всеки две итерации на цикъла x намалява до не повече от  $x/2$  (целочислено деление).
- ▶ Ако броят на итерациите е  $2L$ , то колко е  $L$  ?

# Интересен пример – Евклид (AE)

$$x = 32 = 2^5 = 100000_{(2)}$$

$$100000_{(2)} / 2 = 10000_{(2)}$$

$$10000_{(2)} / 2 = 1000_{(2)}$$

$$1000_{(2)} / 2 = 100_{(2)}$$

$$100_{(2)} / 2 = 10_{(2)}$$

$$10_{(2)} / 2 = 1_{(2)}$$

$$1_{(2)} / 2 = 0_{(2)}$$

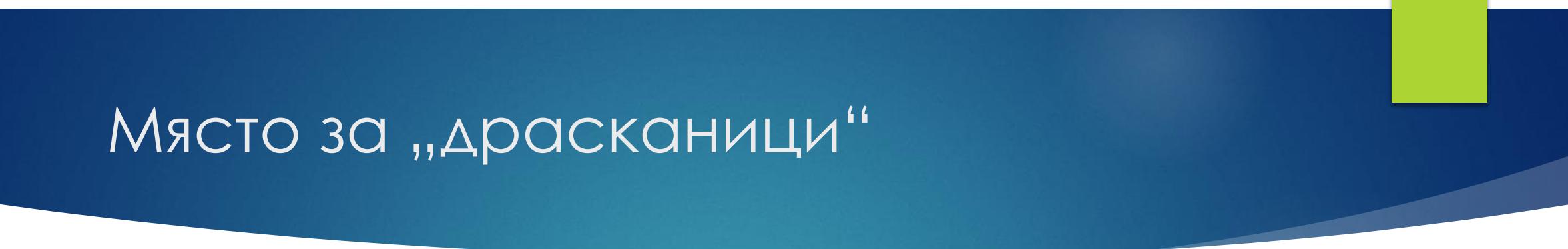
Или  $L = 6 = 5 + 1$  стъпки

1.  $5 = \log_2 2^5$
2. Значи, за да стигнем от  $x$  до 0 с деление на 2 трябват  $\log_2 x$  стъпки;
3. Значи в най-лошия случай AE прави  $2 \cdot \log_2 x$  стъпки и ако вземе за размер на входа  $n = x$ , тогава  $t_{AE}(n) = c \cdot \log_2 n = O(\log_2 n)$ .

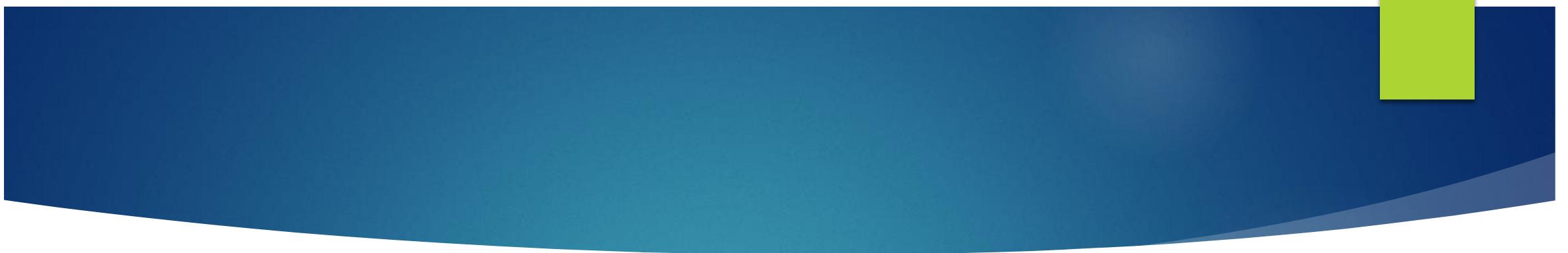
# CSCB039 Алгоритми и програмиране

ЛЕКЦИЯ 7 – КРАЙНИ ГРАФИ И МУЛТИГРАФИ

гл. ас. д-р Слав Емилов Ангелов, НБУ  
проф. Красимир Манев



Място за „драсканици“



# Понятието граф

За дефинирането на графова структура са необходими два вида обекти:

- ▶ Множество  $V = \{v_1, v_2, \dots, v_N\}$  от **върхове**;
- ▶ Множество  $E = \{e_1, e_2, \dots, e_M\}$  от **ребра**;

И някакъв начин, за да се свърже едно ребро  $e_k$  с два от зададените върхове –  $v_i$  и  $v_j$ .

# Понятието граф според вида на ребрата

В зависимост дали ребрата на графовата структура имат ориентация или не различаваме два вида структури:

- Ако двата върха  $v_i$  и  $v_j$  свързани с реброто  $e_k$  образуват наредена двойка  $(v_i, v_j)$  структурата е **ориентирана**;



- Ако двата върха  $v_i$  и  $v_j$  свързани с реброто  $e_k$  образуват ненаредена двойка  $\{v_i, v_j\}$  структурата е **неориентирана**.



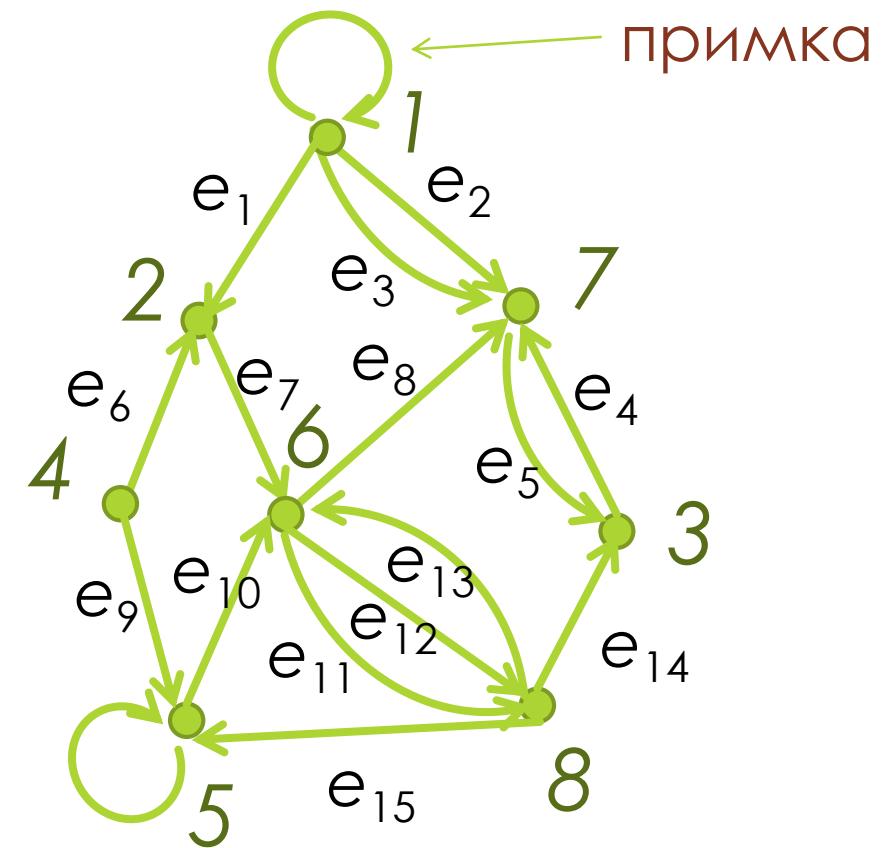
# Понятието граф според броя на ребрата

В зависимост от броя на ребрата свързващи два върха имаме също 2 вида структури:

- ▶ Ако между някои два върха  $v_i$  и  $v_j$  може да има много ребра, то структурата е **мултиграф**;
- ▶ Ако между всеки два върха  $v_i$  и  $v_j$  може да има най-много едно ребро, то структурата е **граф**.

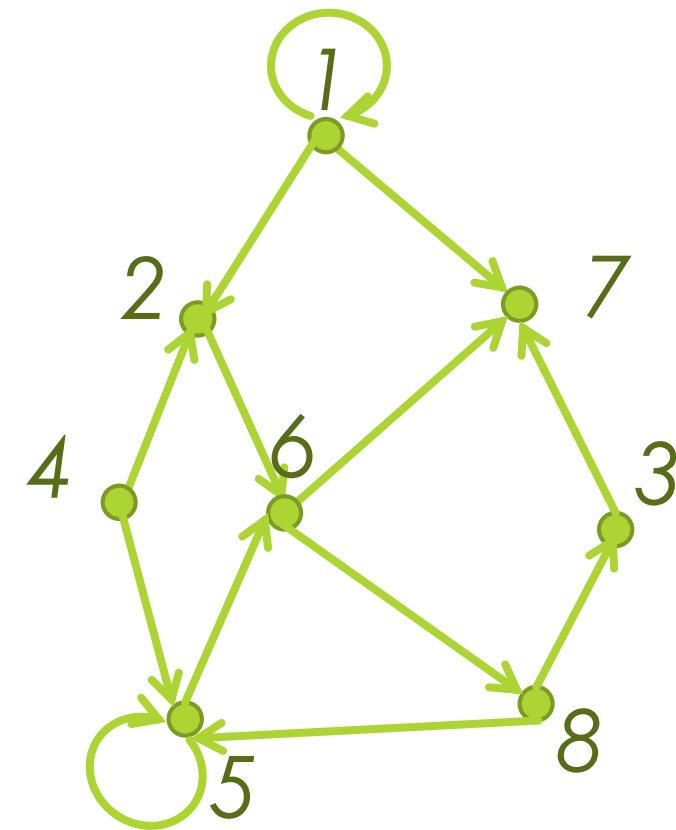
# Ориентиран мултиграф

На фигурата е показан **ориентиран мултиграф (ОМГ)**  $G(V, E, f)$ , където  $V$  е множеството от върхове,  $E$  – множеството от ребра, а функцията  $f: E \rightarrow V \times V$  свързва всяко ребро с два върха.



# Ориентиран граф

Ако функцията  $f$  на ОМГ  $G(V, E, f)$  е  
единозначна ( $e_i \neq e_j \Rightarrow f(e_i) \neq f(e_j)$ )  
тогава структурата се нарича **ориентиран  
граф** – етикетите по ребрата не са  
необходими. Въщност  $E \subseteq V \times V$ .

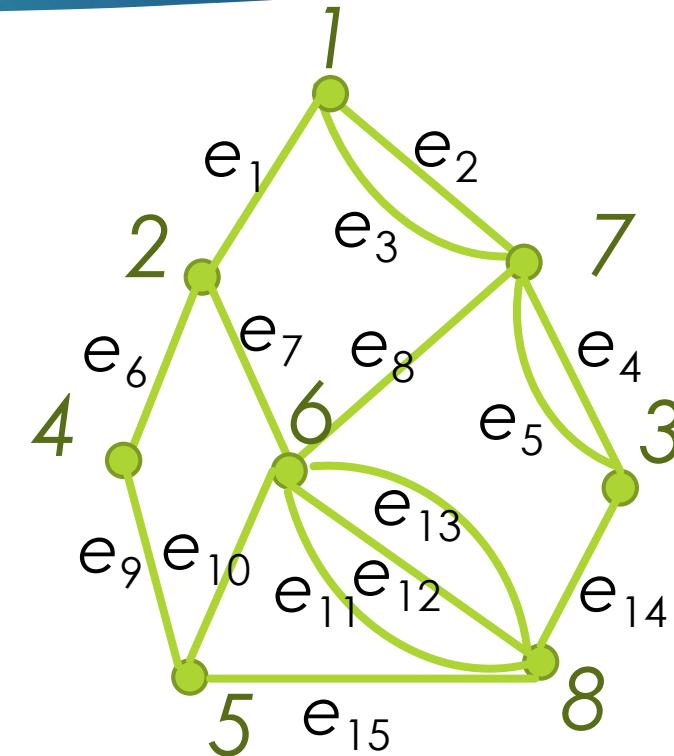


# Неориентиран мултиграф

На фигурата е показан

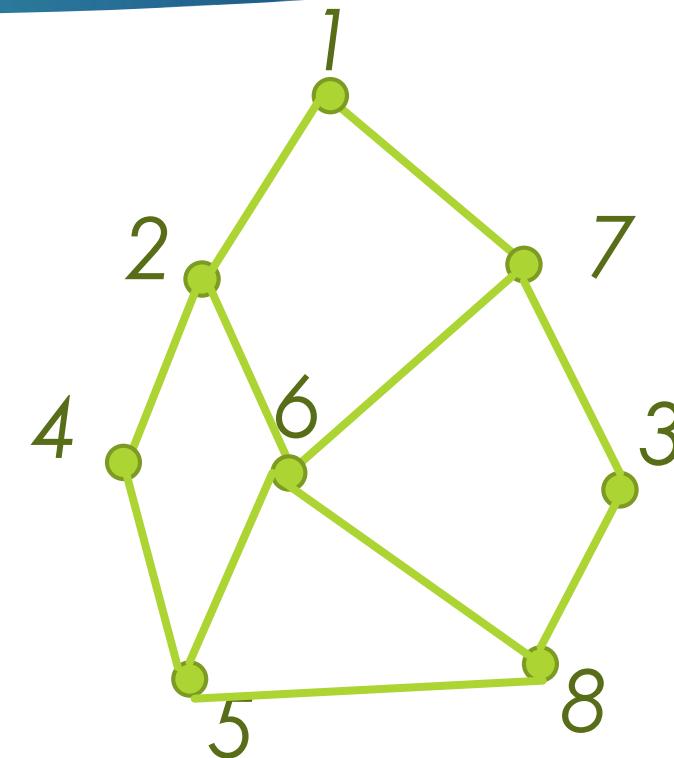
**(неориентиран) мултиграф**

$G(V, E, f)$ , където  $V$  е множеството от върхове,  $E$  – множеството от ребра, а функцията  $f: E \rightarrow 2^V$  свързва с всяко ребро подмножество на  $V$  с 2 върха.



# Неориентиран граф

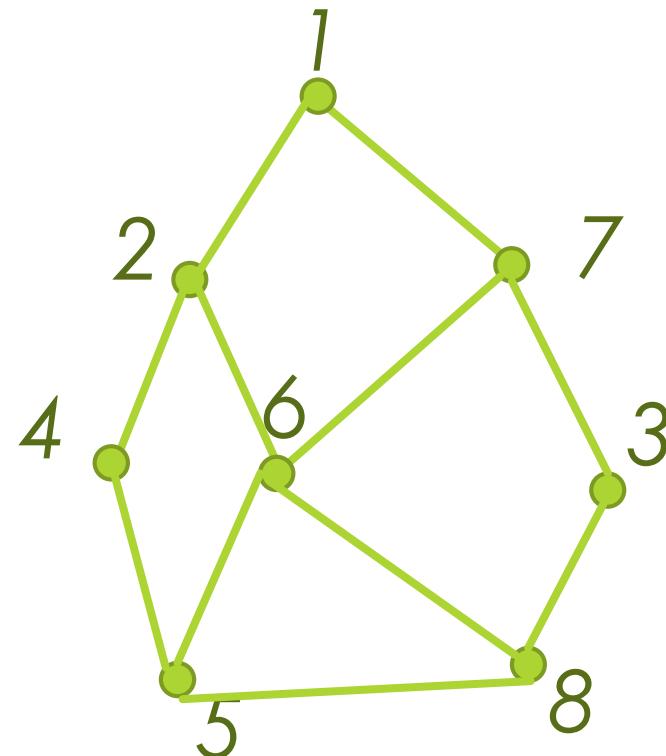
Ако функцията  $f$  на МГ  $G(V, E, f)$  е  
еднозначна ( $e_i \neq e_j \Rightarrow f(e_i) \neq f(e_j)$ )  
тогава структурата е **(неориентиран)  
граф** – етикети по ребрата не са  
необходими. В неориентираните графи  
никога не използваме примки.



# Списък на ребрата

| G  | 1 | 2 |
|----|---|---|
| 1  | 1 | 2 |
| 2  | 1 | 7 |
| 3  | 2 | 4 |
| 4  | 2 | 6 |
| 5  | 3 | 7 |
| 6  | 3 | 8 |
| 7  | 4 | 5 |
| 8  | 5 | 6 |
| 9  | 5 | 8 |
| 10 | 6 | 7 |
| 11 | 6 | 8 |

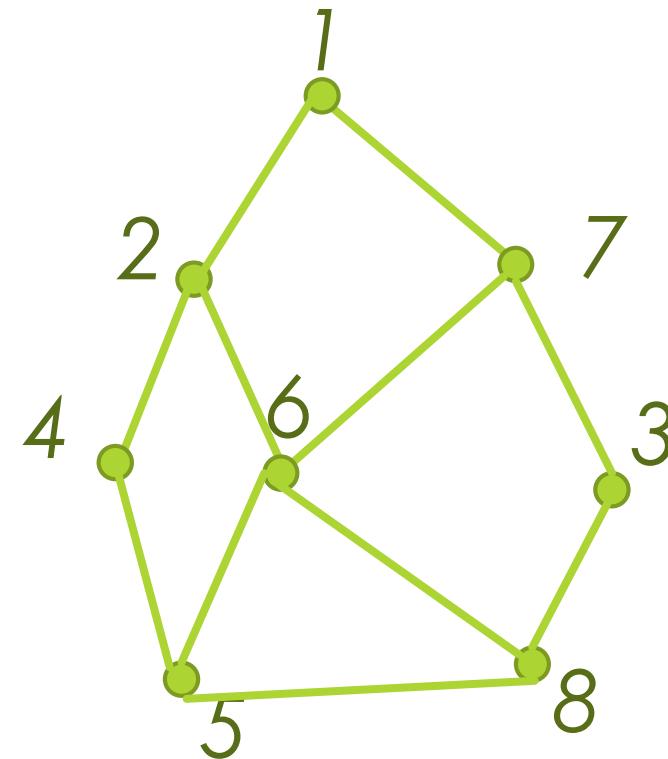
N=8, M=11 и  
(G[i][0], G[i][1]) е i-тото  
ребро на графа



# Матрица на съседства

- $N=8$ ,  $M=11$ , но може да се намери от матрицата;
- $M_{ij}$  = брой ребра от връх  $i$  до връх  $j$ .

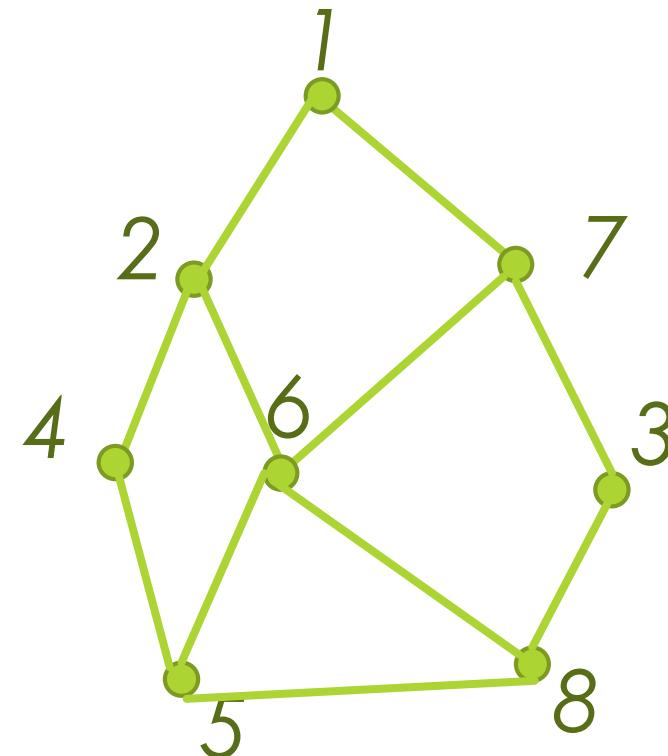
| M | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 2 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 6 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 7 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 8 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |



# Списъци на съседите

- $N=8, M=11$ . Редът  $i$  е списък от съседите на връх  $i$ ;
- $L[i][0]$  – дължина на списъка.

| $L$ | 0        | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|-----|----------|---|---|---|---|---|---|---|---|
| 0   |          |   |   |   |   |   |   |   |   |
| 1   | <b>2</b> | 2 | 7 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2   | <b>3</b> | 1 | 4 | 6 | 0 | 0 | 0 | 0 | 0 |
| 3   | <b>2</b> | 7 | 8 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4   | <b>2</b> | 2 | 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5   | <b>3</b> | 4 | 6 | 8 | 0 | 0 | 0 | 0 | 0 |
| 6   | <b>4</b> | 2 | 5 | 7 | 8 | 0 | 0 | 0 | 0 |
| 7   | <b>3</b> | 1 | 3 | 6 | 0 | 0 | 0 | 0 | 0 |
| 8   | <b>3</b> | 3 | 5 | 6 | 0 | 0 | 0 | 0 | 0 |



# Понятието път

**Маршрут** в ориентиран мултиграф от  $v_{i_0}$  до  $v_{i_L}$  е редицата

$$v_{i_0}, e_{i_1}, v_{i_1}, e_{i_2}, v_{i_2}, \dots, e_{i_L}, v_{i_L}$$

от върхове и ребра, в която

$$f(e_{i_j}) = (v_{i_{j-1}}, v_{i_j}), \quad j=1, 2, \dots, L.$$

- ▶ броят  $L$  на ребрата наричаме **дължина** на маршрута;
- ▶ ако  $v_{i_0} = v_{i_L}$ , тогава маршрутът е **цикъл**.

# Понятието път

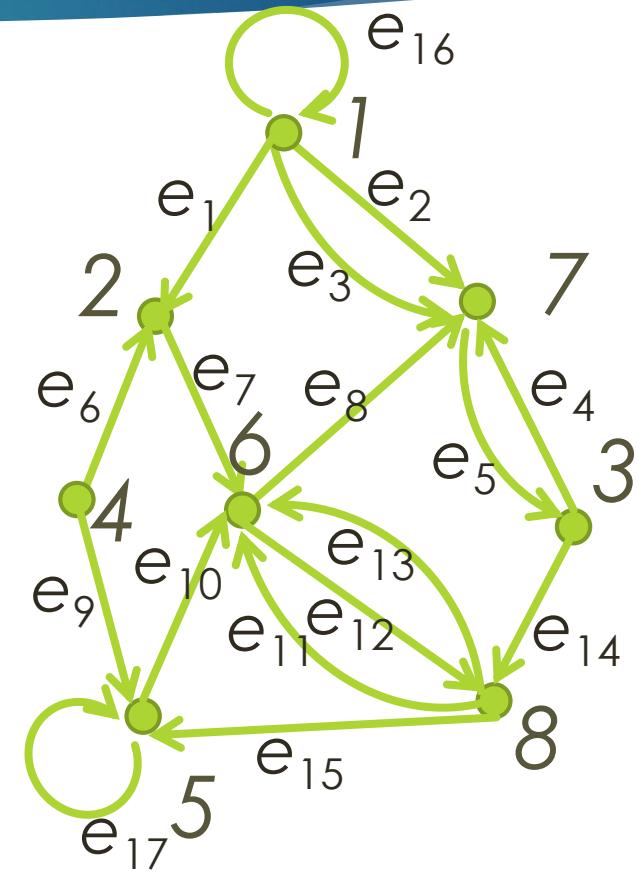
В ОМГ на фигурата

$1, e_{16}, 1, e_2, 7, e_5, 3, e_{14},$   
 $8, e_{13}, 6, e_{12}, 8, e_{11}, 6, e_8, 7$

е маршрут с дължина 8, а

$7, e_5, 3, e_{14}, 8, e_{13}, 6, e_{12}, 8, e_{11}, 6, e_8, 7$

е цикъл с дължина 6.



# Понятието път

**Дефиниция.** **Маршрут в ориентиран граф** от  $v_{i_0}$  до  $v_{i_L}$  е всяка редица

$$v_{i_0}, v_{i_1}, v_{i_2}, \dots, v_{i_L}$$

от върхове такава, че съществува ребро  $e_{i_j}$ , за което

$$f(e_{i_j}) = (v_{i_{j-1}}, v_{i_j}), \quad j=1, 2, \dots, L.$$

- ▶ броят  $L$  на ребрата наричаме **дължина** на маршрута;
- ▶ ако  $v_{i_0} = v_{i_L}$ , тогава маршрутът е **цикъл**.

# Понятието път

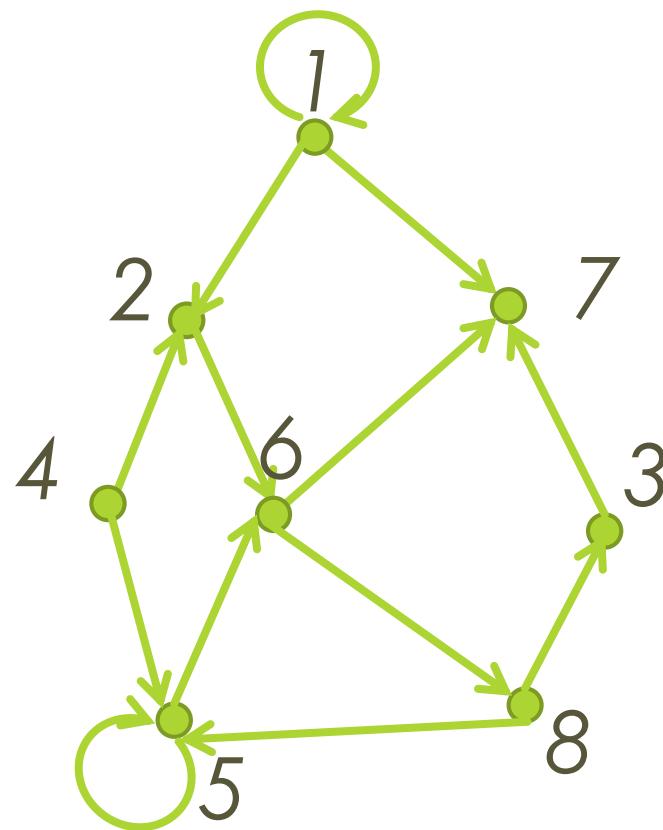
В ориентирания граф на фигурата:

1, 1, 2, 6, 8, 3, 7

е маршрут с дължина 6, а

8, 5, 6, 8, 5, 5, 5, 6, 8

е цикъл с дължина 8.



# Понятието път

Дефиниция.

- A. **Път в неориентиран граф** от  $v_{i_0}$  до  $v_{i_L}$  е всяка редица  
 $v_{i_0}, v_{i_1}, v_{i_2}, \dots, v_{i_L}$  от върхове такава, че  $\exists e_{i_j} = (v_{i_{j-1}}, v_{i_j}), j=1, 2, \dots, n$   
и  $v_{i_{j-1}} \neq v_{i_{j+1}}$ ,  $j=1, 2, \dots, n-1$ . Броят L на ребрата наричаме  
**дължина** на пътя, а ако  $v_{i_0} = v_{i_L}$  тогава пътят е **цикъл**.
- B. Всеки връх  $v_i$  е път с дължина 0 от  $v_i$  до  $v_i$  – **трибуналният път**.

# Понятието път

В графа на фигурата:

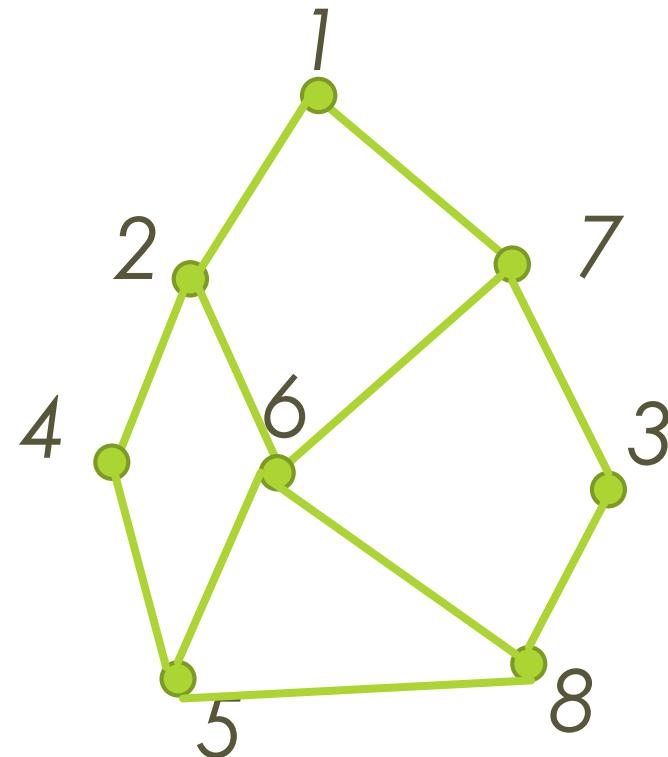
1, 7, 3, 8, 6, 2, 4, 5, 6, 7

е път с дължина 9,

1 и 6 са пътища с дължина 0, а

1, 7, 3, 8, 6, 2, 1

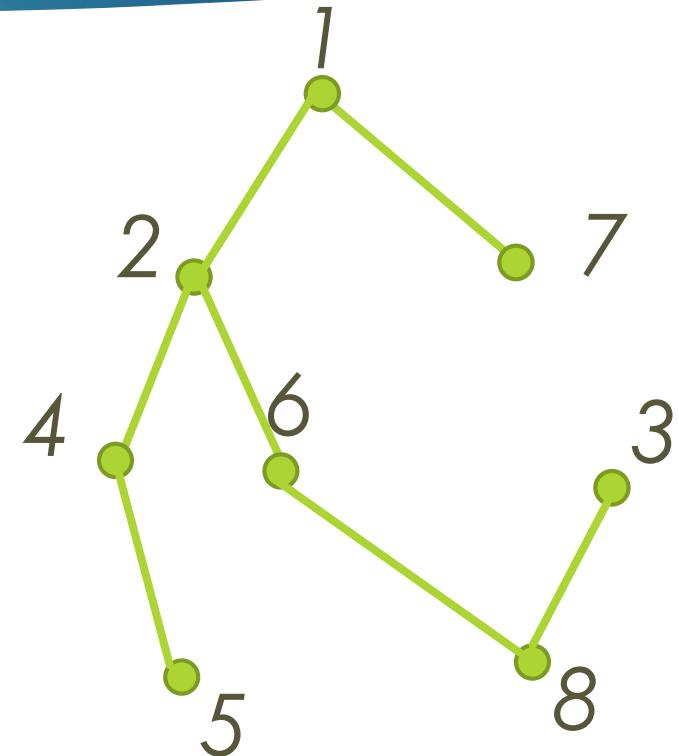
е цикъл с дължина 6.



# Понятието път – важно условие

Условието  $v_{i,j-1} \neq v_{i,j+1}$

е много важно. Без него редицата 1, 2, 4, 2, 1 и дори редицата 1, 2, 1 би трябвало да са цикли. Според нашата дефиниция графът на фигурата, както трябва да бъде, няма цикли.



# Понятието път

Пътят в граф наричаме **прост**, ако не повтаря върхове. Например, пътят

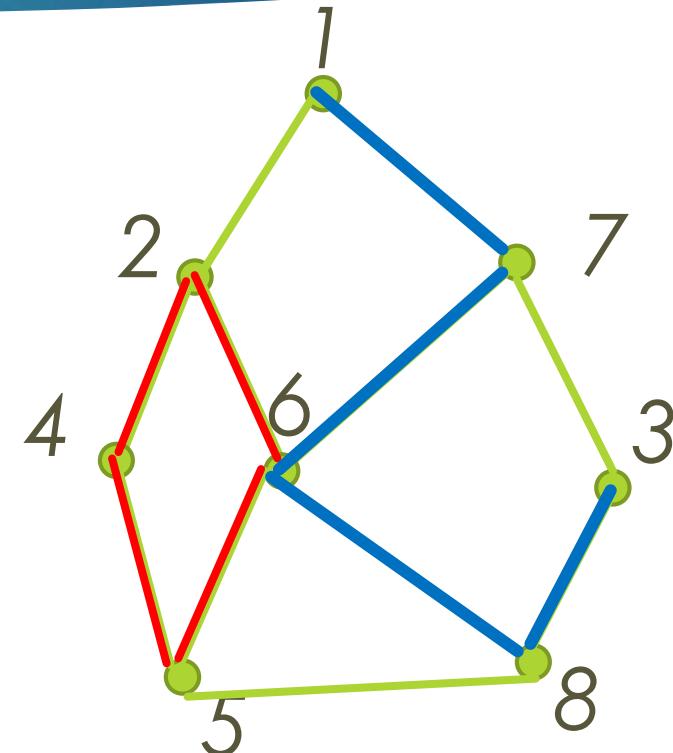
1, 7, 6, 5, 4, 2, 6, 8, 3

не е прост. Но той може да бъде опростен до

1, 7, 6, 8, 3

с изрязване на цикъла

6, 2, 5, 4, 6.



# Понятието път

Дефиницията на **път за неориентиран мултиграф** е не по-различна: редица  $v_{i_0}, e_{i_1}, v_{i_1}, e_{i_2}, v_{i_2}, \dots, e_{i_L}, v_{i_L}$  от върхове и ребра, в която  $f(e_{i_j}) = (v_{i_{j-1}}, v_{i_j})$  и  $e_{i_{j-1}} \neq e_{i_j}$ ,  $j=2, 3, \dots, L$ .

- ▶ Броят  $L$  на ребрата е **дължина на пътя**;
- ▶ Ако  $v_{i_0} = v_{i_n}$  тогава пътят се нарича **цикъл**.

# Понятието път - упражнение

В неориентирания мултиграф на  
фигурата

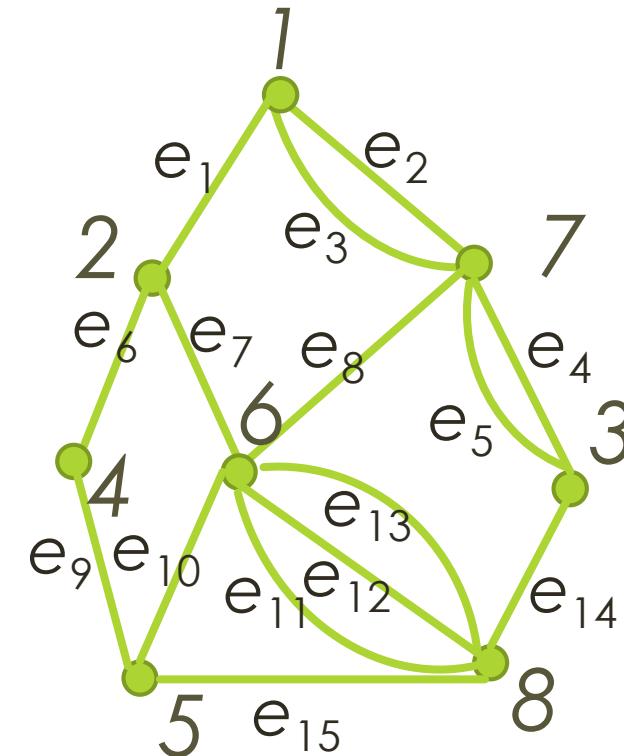
$1, e_2, 7, e_4, 3; \quad 1, e_2, 7, e_5, 3;$

$1, e_3, 7, e_4, 3; \quad 1, e_3, 7, e_5, 3$

са пътища от 1 до 3.

**Намерете друг път от 1 до 3.**

**Колко са пътищата от 1 до 3?**



# MC и пътища в КОМ

Произведение на 2  $n \times n$  матрици  $A = |a_{ij}|$  и  $B = |b_{ij}|$  е  $A \times B = C = |c_{ij}|$ ,

където  $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{in}b_{nj}$  – правилото “скалярно произведение на ред по стълб”.

**Теорема.** Нека  $M^k = |a_{ij}^{(k)}|$  е  $k$ -тата степен на матрицата на съседства  $M$  на ОМГ  $G$ . Тогава  $a_{ij}^{(k)}$  е броят на пътищата от  $v_i$  до  $v_j$  с дължина  $k$ .

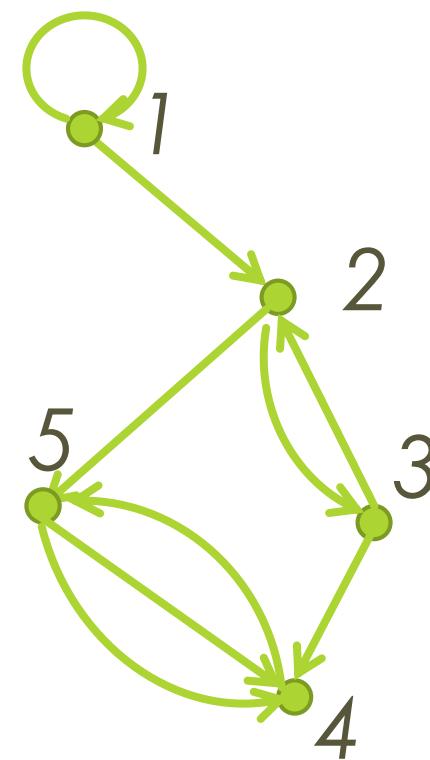
# МС и пътища в КОМ

$M_{ij}$  = брой на ребрата от  $v_i$  до  $v_j$ .

Задача 1: Напишете МС на ОМГ от фигурата.

Задача 2: Намерете броя на пътищата с дължина 4 от връх 1 до 4. А с дължина 5?

|   | 1 | 2 | 3 | 4 | 5 |  |
|---|---|---|---|---|---|--|
| 1 | 1 | 1 | 0 | 0 | 0 |  |
| 2 | 0 | 0 | 1 | 0 | 1 |  |
| 3 | 0 | 1 | 0 | 1 | 0 |  |
| 4 | 0 | 0 | 0 | 0 | 1 |  |
| 5 | 0 | 0 | 0 | 2 | 0 |  |



# МС и пътища в КОМ

Задача: Дадена е МС на ОМГ. Има ли път от даден връх  $u$  до друг връх  $w$  ( $w$  може да съвпада с  $u$ ).

**Лема.** Ако има път в графова структура с  $n$  върха от върха  $u$  до върха  $w$ , тогава има и прост път от  $u$  до  $w$  с дължина  $< n$ .

# Свързаност

**Дефиниция.** Графовата структура  $G$  е *свързана*, ако има път от всеки връх  $u$  до всеки друг връх  $w$ .

**Дефиниция (за ориентирани графи само).** Ориентираната графова структура  $G$  е *слабо свързана*, ако за всеки два различни върха  $u$  и  $w$  съществува път от  $u$  до  $w$  или от  $w$  до  $u$ .

# Подграфи

**Дефиниция.** Нека  $G(V, E)$  е графова структура,  $V' \subseteq V$  и  $E' \subseteq E$  така, че за всяко  $e(u, w) \in E'$ ,  $u \in V'$  и  $w \in V'$ . Тогава  $G'(V', E')$  е подграф на  $G$ .

**Дефиниция.** Ако  $G'(V', E')$  е подграф на  $G(V, E)$ ,  $V' \subseteq V$  и  $E' \subseteq E$  такъв, че за всяко  $e(u, w) \in E$ ,  $u \in V'$  и  $w \in V'$   $\rightarrow e(u, w) \in E'$  казваме, че  $G'$  е породен от  $V'$  подграф на  $G$ .

# Свързани компоненти

**Дефиниция.** Нека  $G(V, E)$  е граф, а  $V_1, V_2, \dots, V_k$  е разбиване на  $V$  такова, че има път между всеки два върха на дадено  $V_i$ ,  $i=1, 2, \dots, k$ , но няма път между никои два върха от два различни дяла на разбиването на  $V$ . Тогава подграфите  $G_1(V_1, E_1)$ ,  $G_2(V_2, E_2), \dots, G_k(V_k, E_1)$ , породени от  $V_1, V_2, \dots, V_k$  наричаме свързани компоненти на  $G$ .

# Задачи

- 1. Краен неориентиран граф е зададен със списък на ребрата.  
Напишете програма, която въвежда графа и го извежда, представен с  
матрица на съседствата.**
  
- 2. Краен неориентиран граф е зададен със списък на ребрата.  
Напишете програма, която въвежда графа и го извежда, представен  
със списъци на съседите.**

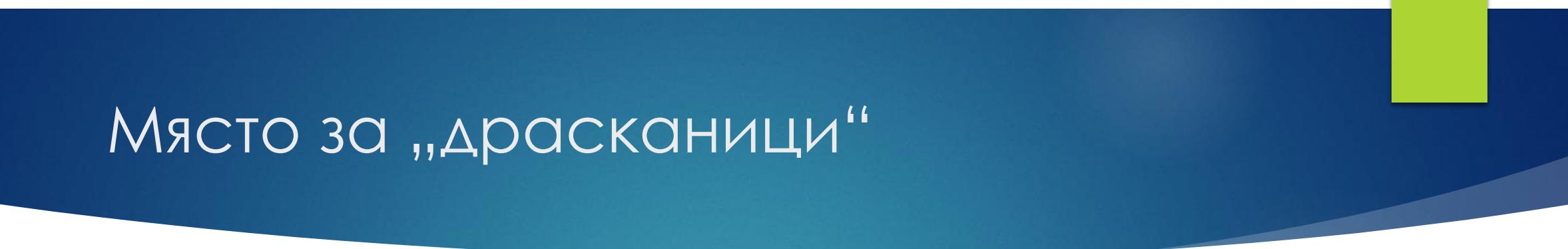
# Задачи

- 3. Решете задачи 1 и 2 за неориентиран мултиграф.**
- 4. Решете задачи 1 и 2 за ориентиран граф.**
- 5. Решете задачи 1 и 2 за ориентиран мултиграф.**

# CSCB039 Алгоритми и програмиране

ЛЕКЦИЯ 8

гл. ас. д-р Слав Емилов Ангелов, НБУ  
проф. Красимир Манев



Място за „драсканици“

# Обхождане на графи

- ▶ Под **обхождане на граф** разбираме някакво систематично, подчинено на зададени правила, посещение на върховете и/или ребрата на графа;
- ▶ Някои обхождания могат да се използват като **алгоритмични схеми**, за решаване по еднотипен начин на различни задачи.

# Алгоритмични схеми

Това е възможността по сходен начин да бъдат решавани различни задачи с еднотипни алгоритми. Алгоритмичната схема **не е алгоритъм**, макар много да прилича на такъв, защото не решава конкретна задача, а **може да се настройва според задачата**, често не много трудно, и дава бърз начин за построяване на съответен алгоритъм.

Примери:

- ▶ Сливане на два сортирани масива;
- ▶ Разделяй и владей;
- ▶ Обхождане на граф в ширина;
- ▶ Обхождане на граф в дълбочина.

Забележка: Понятието алгоритмична схема е дефинирано само в някои държави.

# Обхождане на графи

Популярни обхождания на графи от общ тип са:

- ▶ Ойлеровото обхождане,
- ▶ Хамилтоновото обхождане,
- ▶ Обхождането в ширина,
- ▶ Обхождането в дълбочина.

# ВНИМАНИЕ !!!

**Поради имплементациите и използваните структури от данни,  
классически подход е върховете на графи да се номерират от  
индекс 1, не то 0 !!!**

# Схемата „Обхождане в ширина“

Даден е свързан граф  $G(V, E)$

Основни понятия:

- ▶ **Начален връх**  $r$ ;
- ▶ **Нива** на обхождането в ширина:

$L_0, L_1, L_2, \dots, L_k$  – разбиване на  $V$ ,

$$L_i \cap L_j = \emptyset, \quad i \neq j,$$

$$L_0 \cup L_1 \cup L_2 \cup \dots \cup L_k = V;$$

- ▶ **U – обходените** върхове .

# Схемата „Обхождане в ширина“

Процедура:

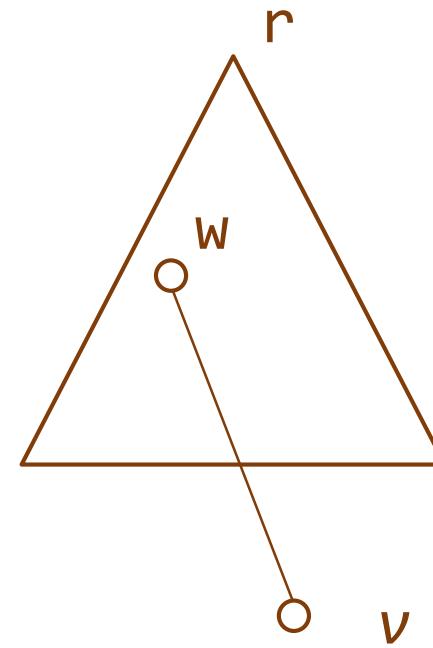
1.  $i = 0; L_0 = U = \{r\};$
2. Ако  $U = V$ , КРАЙ;
3. **Построяваме** следващо ниво:  
 $L_{i+1} = \{ v \mid v \notin U, \exists w \in L_i, (v, w) \in E \};$
4.  $\forall v \in L_{i+1} \rightarrow U = U \cup \{v\};$
5.  $i++;$  премини към 2.

# Дърво и кореново дърво

**Дефиниция.** *Дърво* е свързан граф без цикли. - неудобна

**Дефиниция.** *Кореново дърво (КД)* - корен *r*.

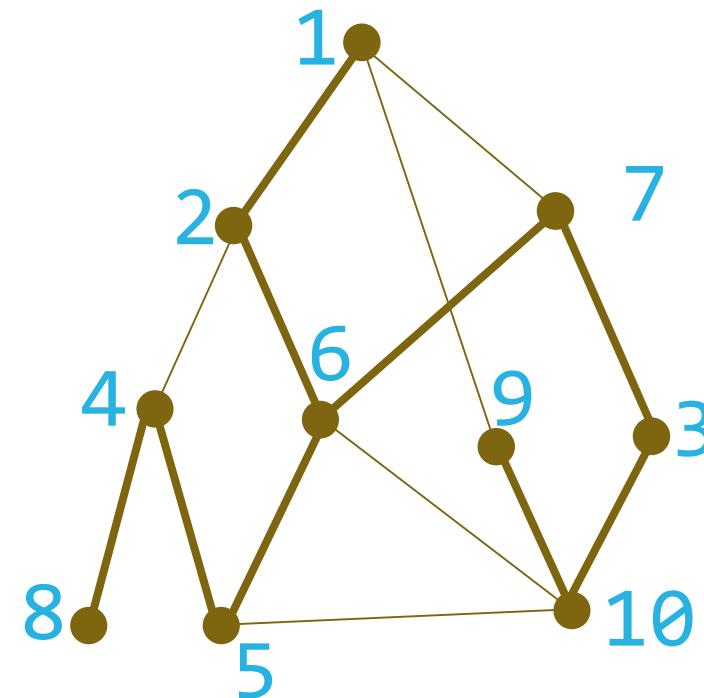
- a.  $T(\{r\}, \emptyset)$  е КД.
- b. Ако  $T(V, E)$  е КД и имаме  $w \in V, v \notin V$ , тогава
- c.  $T'(\ V \cup \{v\}, \ E \cup \{(w, v)\} \ )$  също е КД.



# Покриващо дърво

**Дефиниция.** Даден е свързан граф  $G(V, E)$ . Дърво  $T(V, E')$  такова, че  $E' \subseteq E$ , ако има такова, наричаме **покриващо дърво (ПД)** на  $G$ .

**Задача:** По зададен граф  $G(V, E)$  да се построи ПД на  $G$ , ако има такова.



# Покриващо дърво

Съществуването на покриващо дърво в свързан граф не е очевидно.

**Теорема.** Графът  $G(V, E)$  е свързан т.с.т.к. има ПД. (НДУ)

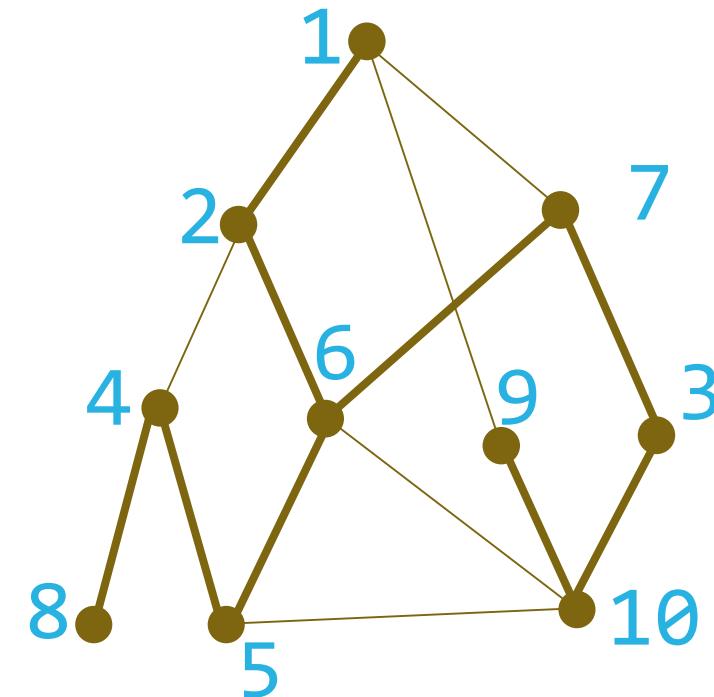
**Доказателство:**

1. Достатъчност – очевидна;
2. Необходимост

while( $G$  има цикъл)

```
{ премахни ребро  
от цикъла
```

```
}
```

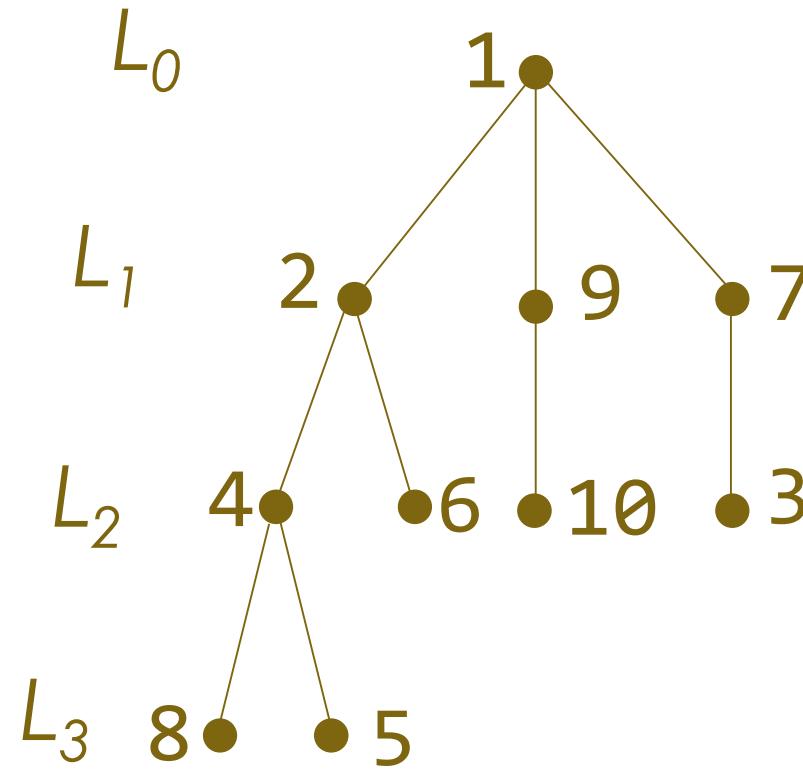
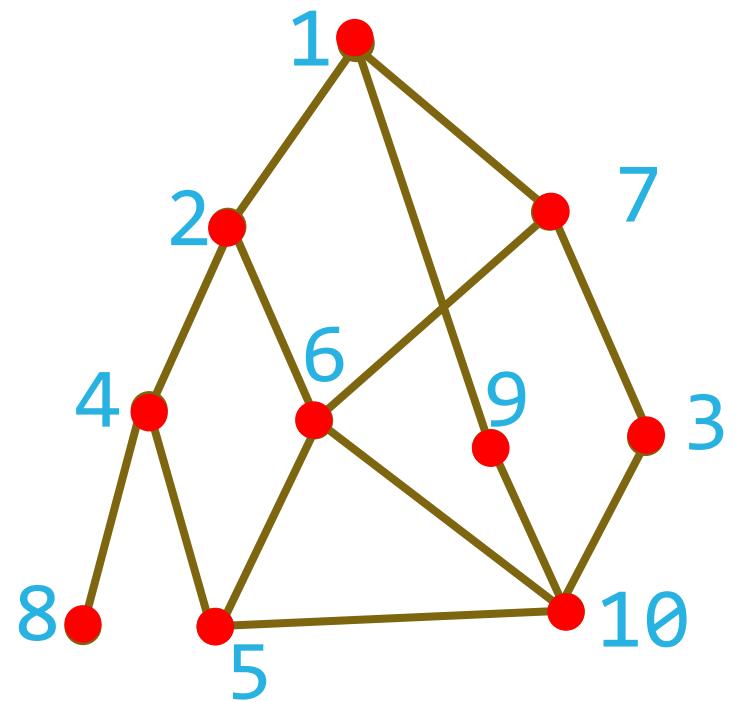


# Алгоритъм за построяване ПД с „обхождане в ширина“

Процедура:

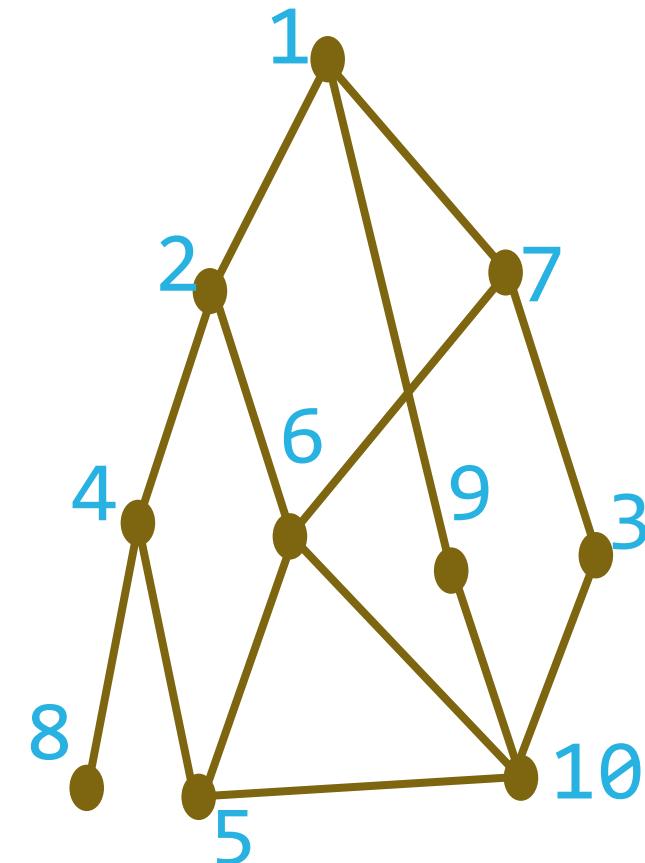
1.  $i=0; L_0=U=\{r\}; T(U, E=\emptyset);$
2. Ако  $U = V$ , КРАЙ;
3. Построяваме  $L_{i+1}=\{ v \mid v \notin U, \exists w \in L_i, (v, w) \in E \};$
4.  $\forall v \in L_{i+1} \rightarrow U=U \cup \{v\}; E= E \cup \{(w, v)\};$
5.  $i++;$  премини към 2.

# Алгоритъм за построяване ПД с „обхождане в ширина“



# Представяне на граф със списъци на съседите

|    | 0 |   |    |    |   |   |   |   |   |  |
|----|---|---|----|----|---|---|---|---|---|--|
| 1  | 3 | 2 | 9  | 7  | 0 | 0 | 0 | 0 | 0 |  |
| 2  | 3 | 1 | 6  | 4  | 0 | 0 | 0 | 0 | 0 |  |
| 3  | 2 | 7 | 10 | 0  | 0 | 0 | 0 | 0 | 0 |  |
| 4  | 3 | 2 | 5  | 8  | 0 | 0 | 0 | 0 | 0 |  |
| 5  | 3 | 4 | 6  | 10 | 0 | 0 | 0 | 0 | 0 |  |
| 6  | 4 | 2 | 7  | 10 | 5 | 0 | 0 | 0 | 0 |  |
| 7  | 3 | 1 | 6  | 3  | 0 | 0 | 0 | 0 | 0 |  |
| 8  | 1 | 4 | 0  | 0  | 0 | 0 | 0 | 0 | 0 |  |
| 9  | 2 | 1 | 10 | 0  | 0 | 0 | 0 | 0 | 0 |  |
| 10 | 4 | 5 | 6  | 9  | 3 | 0 | 0 | 0 | 0 |  |



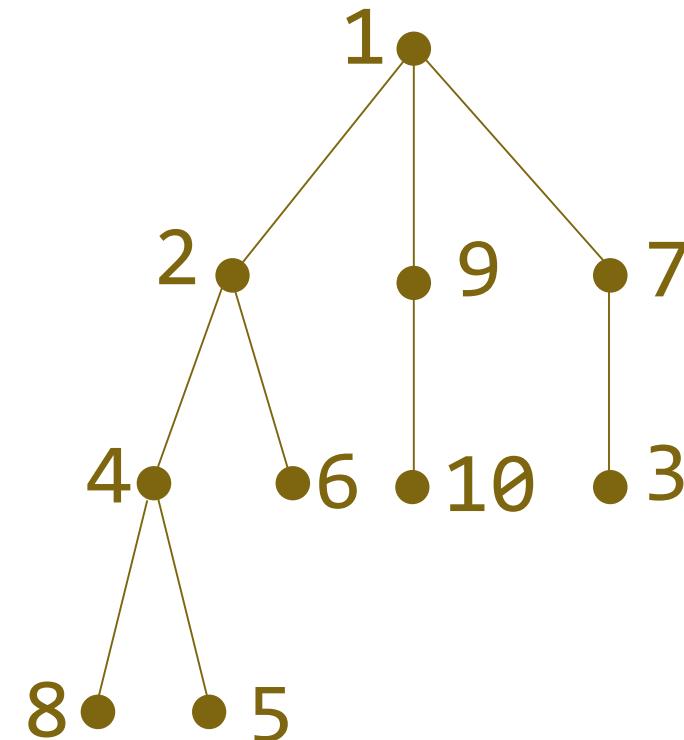
# Превръщане на „CP“ в „CC“

```
int N,M,G[MAXN][MAXN];  
  
int main()  
{  int i,u,v;  
    . . .  
    scanf("%d %d", &N, &M);  
    . . .  
    for(i=1;i<=N;i++) G[i][0]=0;  
    . . .  
    for(i=1;i<=M;i++) {  
        scanf("%d %d",&u,&v);  
        G[u][++G[u][0]]=v;  
        G[v][++G[v][0]]=u;  
    }    . . .  
}
```



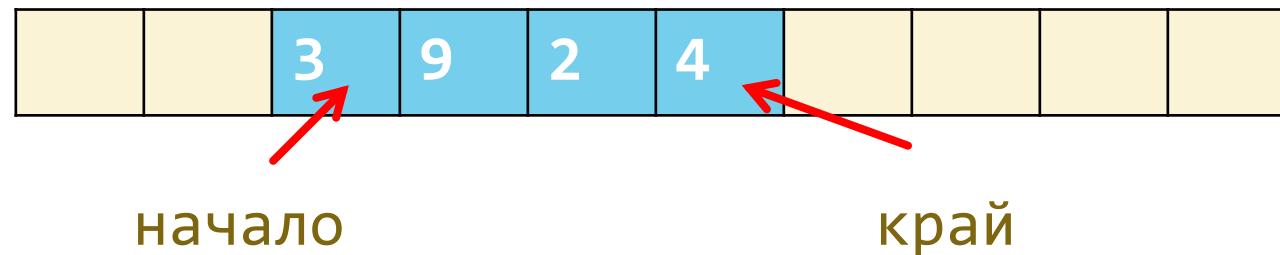
# Представяне на кореново дърво със списък на бащите

| i  | P[i] |
|----|------|
| 1  | 0    |
| 2  | 1    |
| 3  | 7    |
| 4  | 2    |
| 5  | 4    |
| 6  | 2    |
| 7  | 1    |
| 8  | 4    |
| 9  | 1    |
| 10 | 9    |



# Опашка

Редица от еднотипни данни, с **начало** и **край**. Новите елементи се добавят в края, а изваждането става от началото (**first in-first out**):



Операции:

- `void make_empty()` – празна опашка
- `void push(int x)` - добавя елемент
- `int pop()` – изважда елемент
- `bool not_empty()` – дали е празна

# Опашка

```
int Q[MAXN], b, e;  
void make_empty(){b=0;e=-1;}  
void push(int x){Q[++e]=x;}  
int pop(){return Q[b++];}  
bool not_empty(){return b<=e;}
```

**Отбелязването на  $v$  като обходен:**

```
int U[MAXN]; // мар  
for(i=1;i<=N;i++) U[i]=0;  
... U[i]=1; ... // i е обходен
```

# Алгоритъм за построяване ПД с „обхождане в ширина“ чрез опашка

## Процедура

1.  ~~$i=0; L_0=U=\{r\}; T(\{r\}, \emptyset); U[r]=1; P[r]=0;$~~

поставяме  $r$  в празна опашка

2. Ако  ~~$U = V$~~ , край

докато опашката е непразна {

3. Постр.  ~~$L_{i+1} = \{v \mid v \notin U, \exists w \in L_i, (v, w) \in E\}$~~

4.  ~~$\forall v \in L_{i+1} \rightarrow U = U \cup \{v\};$~~

$x = \text{pop}(); \forall y (x, y) \in E, U[y] == 0 \{ \text{push}(y);$

$U[y] = 1; E = E \cup \{(x, y)\}; P[y] = x; \}$

5.  ~~$i++;$~~  премини към 2.

# Имплементация за построяване ПД с „обхождане в ширина“ чрез опашка

```
int N,M;  
  
int U[MAX+1],Q[MAX],P[MAX],G[MAX][MAX];  
void BFS(int r){  
  
    int x,y,i;  
  
    for(i=1;i<=N;i++) U[i]=0;  
    make_empty(); push(r);  
    U[r]=1;P[r]=0;  
    ...  
    }  
  
    while(not_empty()){  
        x=pop();  
        for(i=1;i<=G[x][0];i++){  
            y=G[x][i];  
            if(!U[y]){  
                push(y); U[y]=1;  
                P[y]=x;  
            }  
        }  
    }  
}
```

# Задачи

1. Въведете функциите, имплементиращи опашка и ги тествайте.
2. Въведете функцията BFS.
3. Напишете главна програма, която да въвежда броя на върховете N, броя на ребрата M и графа в представяне Списъци на съседите, след което извиква BFS да построи ПД на графа.
4. Тествайте програмата с примера от лекцията.

# Още задачи

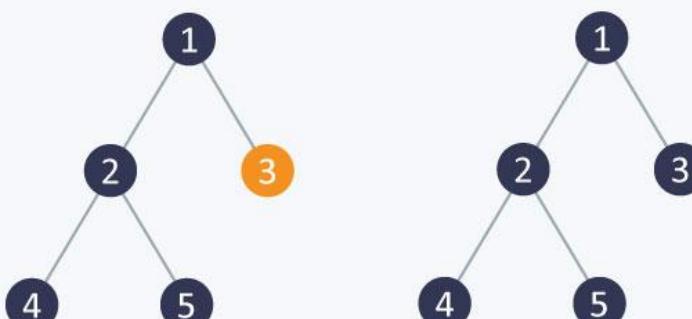
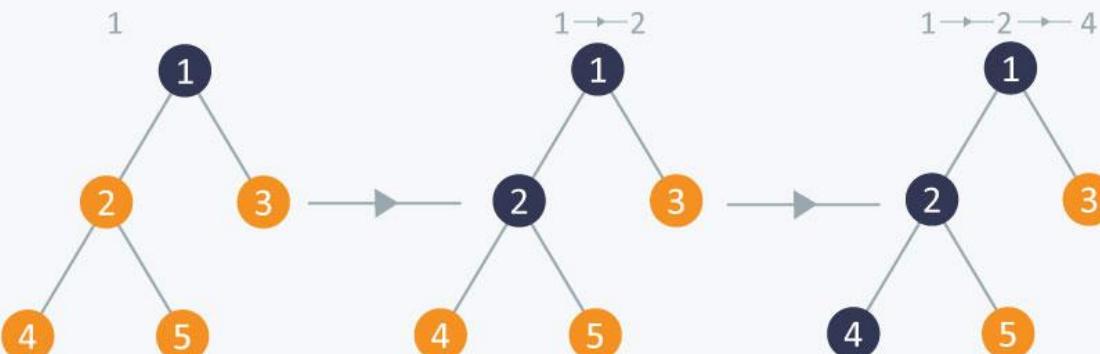
Задача 1. Каква е сложността по време на функцията `BFS()`?

Задача 2. Напишете програма, която проверява дали зададен граф е свързан.

Задача 3. Напишете програма, която построява гора от покриващи дървета на несвързан граф.

# „Обхождане в дълбочина“

DFS



1 → 2 → 4 → 5

1 → 2 → 4 → 5 → 3

# Схемата „Обхождане в дълбочина“

Даден е свързан граф  $G(V, E)$ :

Основни понятия:

- ▶ **Начален** връх  $r$
- ▶ **Текущ** връх  $t$
- ▶  $\forall v \in V, v \neq r$  означаваме  $p(v)$  да е **родител** на  $v$  при обхождането
- ▶  $U$  – множеството на **обходените** върхове

# Схемата „Обхождане в дълбочина“

## Процедура

1.  $t = r; U = \{r\}$

2. Търсим  $v \notin U, (t, v) \in E$

3. Ако има такъв

$p(v)=t; U=U \cup \{v\}; t=v$ ; Отиди на 2.

4. Иначе

4.1. Ако  $t=r$ , КРАЙ.

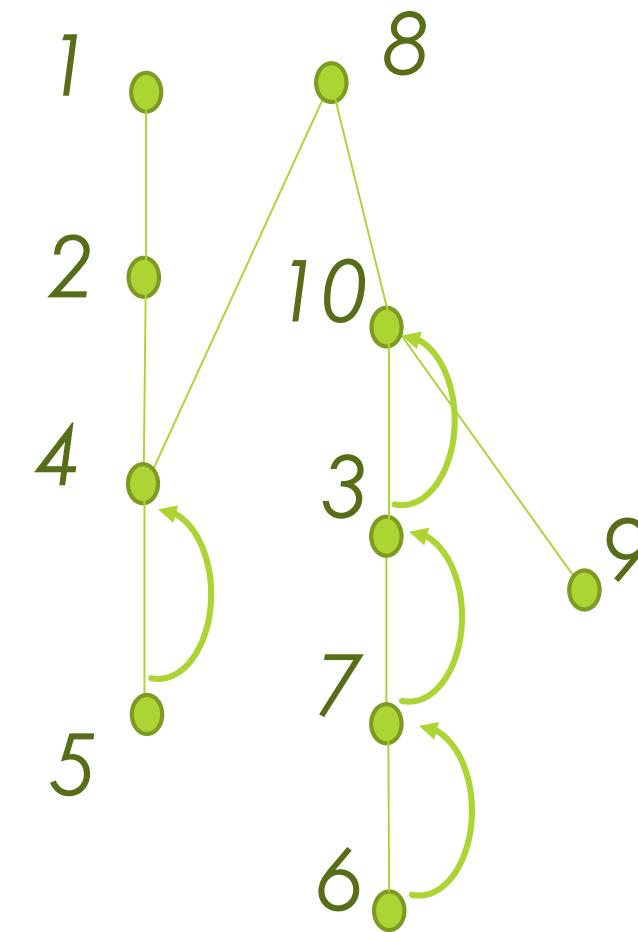
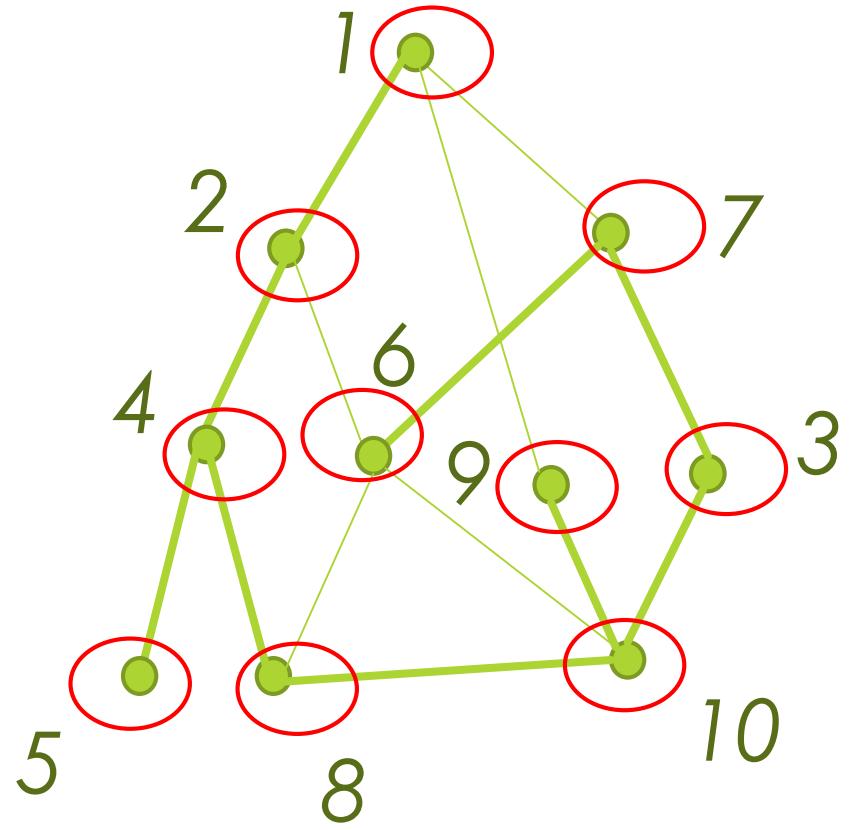
4.2. иначе  $t=p(t)$ ; Отиди на 2.

# Алгоритъм за построяване на ПД „в дълбочина“

## Процедура:

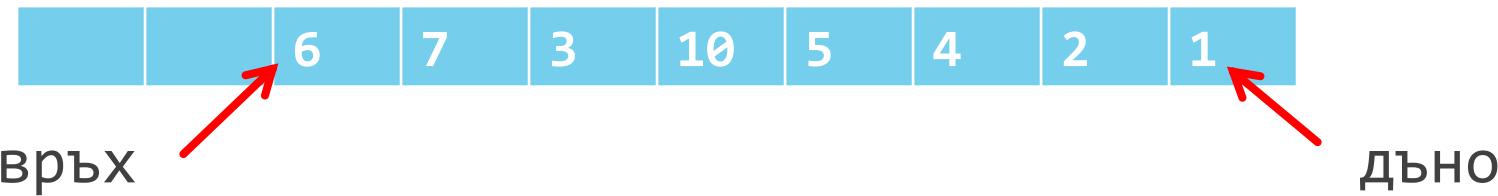
1.  $t = r; U = \{r\}; T(U, E1=\emptyset)$
2. Търсим  $v \notin U, (t, v) \in E$
3. **Ако** има такъв  
 $p(v)=t; E1=E1 \cup \{(t, v)\}; U=U \cup \{v\}; t=v;$   
отиди на стъпка 2.
4. **Иначе**
  - 4.1. Ако  $t=r$ , край.
  - 4.2. Иначе  $t=p(t)$ ; отиди на стъпка 2.

# Алгоритъм за построяване ПД с „обхождане в дълбочина“



# Стек

**Редица от еднотипни данни, с връх и дъно. Новите елементи се добавят над върха, а изваждането става от върха (first in – last out):**



Операции:

- void empty\_stack() – празен стек
- void push(int x) - добавя елемент
- void pop() – изважда елемент
- int look() – показва върха
- bool not\_empty() – дали е непразен

# Стек

```
int S[MAXN],top;
```

- void empty\_stack(){top=-1;}
- void push(int x){S[++top]=x;}
- void pop(){top--;}
- int look(){return S[top];}
- bool not\_empty(){return top>-1;}

**А ако започнем от 0 ?**

# Алгоритъм за построяване на ПД „в дълбочина“ със стек

Процедура:

1.  ~~$t=r$ ; make\_empty(); push( $r$ );  
 $U=\{r\}$ ;  $U[r]=1$ ;  $E=E-\emptyset$   $P[r]=0$ ;~~  
~~while(not\_empty()) {t=look();~~
2. ~~Търсим  $v \in U, (t, v) \in E$~~
3. Ако има такъв ~~if( $G[t][0]>0$ ) {  
int  $v=G[t][G[t][0]-1]$ ; if(! $U[v]$ )  
 $E=E \cup \{(t, v)\}; U=U \cup \{v\}; P(v)=t;$   
 $U[v]=1; P[v]=t; t=v$ ; отиди на 2 push(v);}~~
4. ~~Ако няма : Ако  $t=r$ , край.~~  
Иначе  ~~$t=p(t)$ ; отиди на стъпка 2. else pop();}}~~

# Алгоритъм за построяване ПД с „обхождане в дълбочина“

```
int U[MAXN], S[MAXN], P[MAXN], G[MAXN][MAXN], N;  
  
void DFS (int r){  
    for(int i=1;i<=N;i++) U[i]=0;  
    empty_stack();  
    push(r); U[r]=1; P[r]=0;  
    while(not_empty()){ int t=look();  
        if(G[t][0]>0){  
            int v=G[t][G[t][0]--];  
            if(!U[v]) {U[v]=1; P[v]=t; push(v);}  
        } else pop();  
    } }
```

Какво е това MAXN ?  
Колко голямо трябва да е?

# Задачи

1. Въведете функцията DFS.
2. Напишете главна програма, която да въвежда броя на върховете  $N$ , броя на ребрата  $M$  и графа в представяне със списъци на съседите, след което извиква DFS да построи ПД на графа.
3. Тествайте програмата с примера от лекцията.

# Още задачи

Задача 1. Каква е сложността по време на функцията `DFS()`?

Задача 2. Напишете програма, която проверява дали зададен граф е свързан с обхождане в дълбочина.

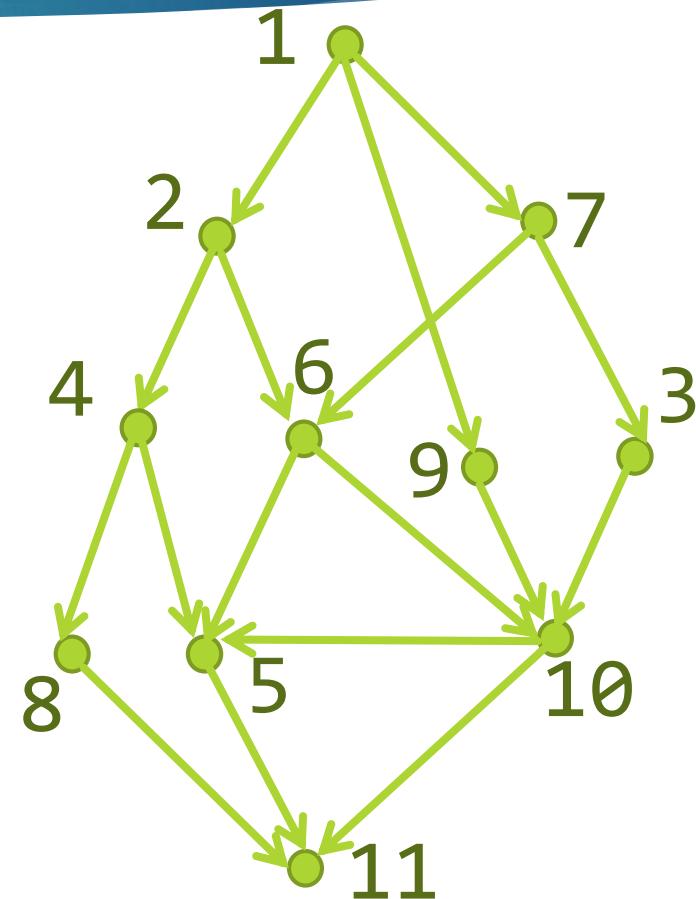
Задача 3. Напишете програма, която построява гора от покриващи дървета на несвързан граф с обхождане в дълбочина.

# Ориентиран ацикличен граф

**Ориентиран ацикличен граф (dag)**

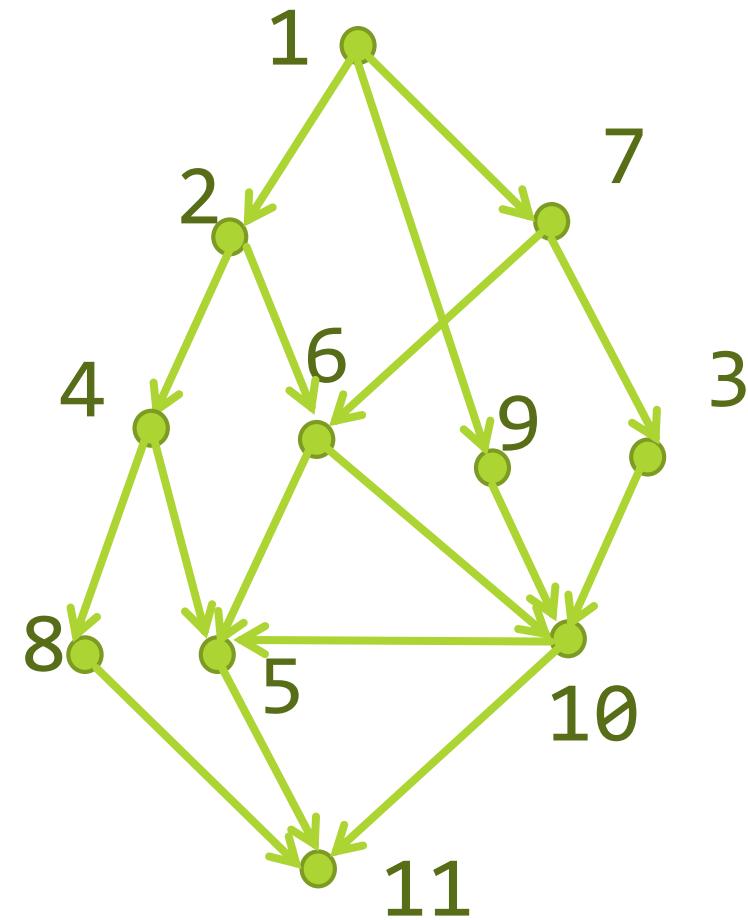
е ориентиран граф без цикли.

Моделира различни ситуации от живота.  
Например, отделните работи на един  
проект, като ориентираните ребра  
задават предхождане на работите.



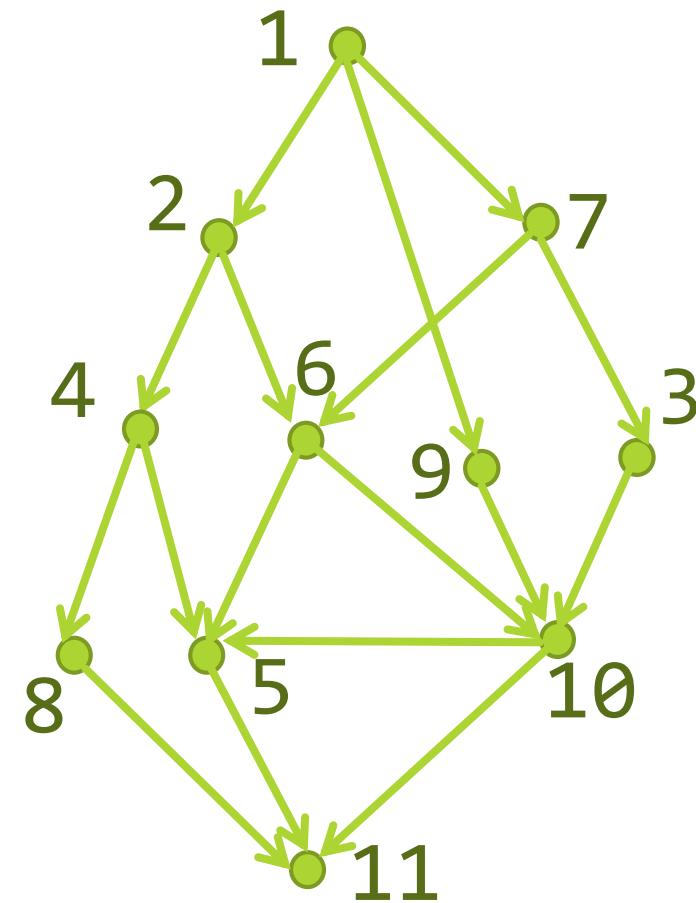
# Топологическо сортиране

Върховете на dag могат да се подредят в редица така, че ако има път от  $v$  до  $w$ , то  $v$  да е преди  $w$  в редицата – **топологическо сортиране**.



# Представяне на даг със списъци на децата

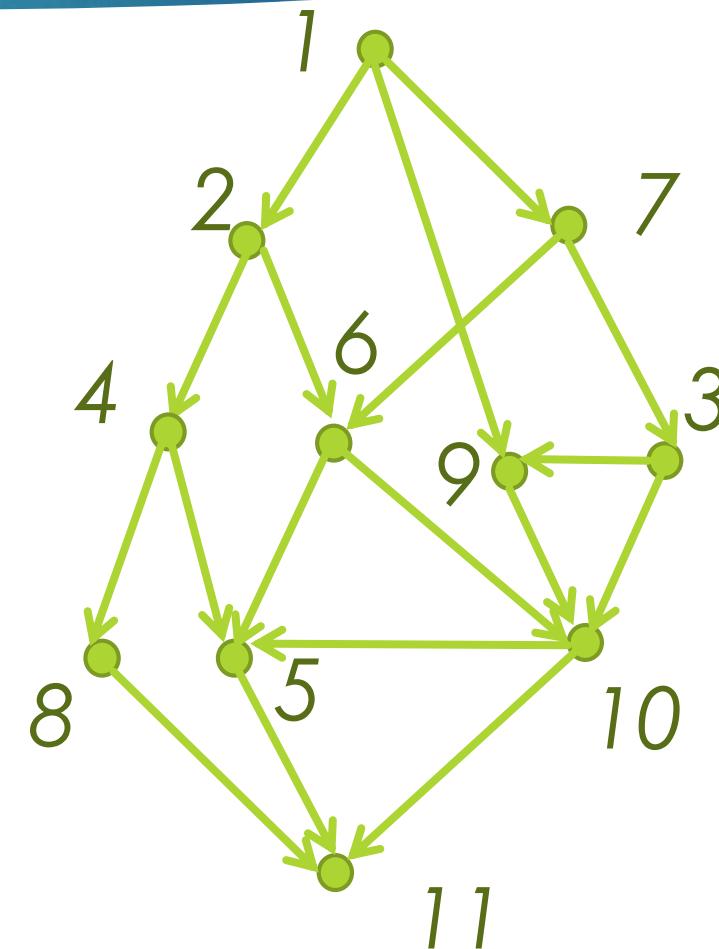
| 1  | 3 | 2  | 9  | 7 | 0 | 0 | 0 | 0 | 0 | 0 |
|----|---|----|----|---|---|---|---|---|---|---|
| 2  | 2 | 6  | 4  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3  | 1 | 10 | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4  | 2 | 5  | 8  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5  | 1 | 11 | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6  | 2 | 5  | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7  | 2 | 6  | 3  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8  | 1 | 11 | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9  | 1 | 10 | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 2 | 5  | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0  | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |



# Топологическо сортиране

**Теорема.** При обхождане в дълбочина на dag върховете напускат стека в ред **обратен** на едно топологическо сортиране.

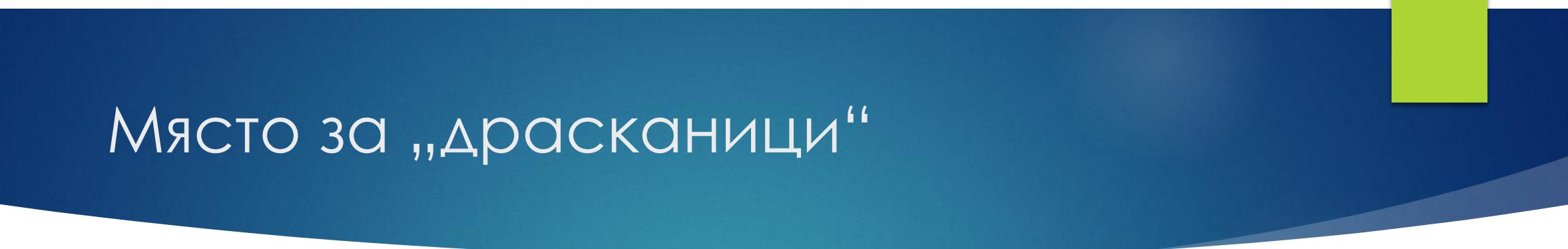
|   |   |   |   |   |   |    |   |   |   |    |
|---|---|---|---|---|---|----|---|---|---|----|
| 1 | 7 | 3 | 9 | 2 | 6 | 10 | 4 | 5 | 8 | 11 |
|   |   |   |   |   |   |    |   |   |   |    |



# CSCB039 Алгоритми и програмиране

ЛЕКЦИЯ 9

гл. ас. д-р Слав Емилов Ангелов, НБУ  
проф. Красимир Манев



Място за „драсканици“

# Въведение (проф. Манев)

Темата «Най-къс път в граф» е една от най-важните алгоритмични задачи не само в алгоритмичната теория на графите, но и изобщо в алгоритмиката, с многообразни практически приложения. В лекцията са описани три различни задачи за намиране на най-къс път в граф, но за практиката най важната е "от един връх да всички останали". Причината за това е съществуването на структура, която наричаме Дърво на най-къси пътища от един връх до всички останали. Разгледани са два характерни случая - на граф без тегла по ребрата и на граф с тегла по ребрата. В първия случай, алгоритъмът се състои в построяване на покриващо дърво в ширина започвайки от началния връх - задача, която вече познаваме от предишна лекция (припомнете си този алгоритъм), а във втория - много важният "Алгоритъм на Дейкстра". Този алгоритъм е особено важен, защото по същество е алгоритмична схема (релаксационната схема), по която могат да се строят и други алгоритми - за най-пропускливи пътища, за най-надежден път и т.н. Темата завършва с много прост и надежден алгоритъм (който сам по себе си е също алгоритмична схема) - "Алгоритъмът на Флойд" за намиране на най-къс път от всеки връх до всеки друг връх за не много големи графи, тъй като сложността му е  $O(N^3)$ .

# СЛОЖНОСТЬ НА BFS ?

```
int N,M;  
int U[MAX+1],Q[MAX],P[MAX],G[MAX][MAX];  
void BFS(int r){  
    int x, y, i;  
    for(i=1; i<=N; i++) U[i]=0;  
    make_empty(); push(r);  
    U[r]=1;P[r]=0;  
    ...  
    while(not_empty()){  
        x=pop();  
        for(i=1;i<=G[x][0];i++){  
            y=G[x][i];  
            if(!U[y]){  
                push(y); U[y]=1;  
                P[y]=x;  
            }  
        }  
    }  
}
```

# Сложност на BFS

Сложността е  $O(|V| + |E|)$ , но дори и графът да не е мултиграф, може  $|E| \leq |X^2|$ .

Можем ли да представим сложността чрез МПДП? Колко ще бъде тя?

# Други въпроси

Ще работи ли BFS за неориентирани мултиграфи?

А за ориентирани? Трябва ли да пипнем нещо по имплементацията от предните слайдове?

# СЛОЖНОСТЬ на DFS ?

```
int U[MAXN], S[MAXN], P[MAXN], G[MAXN][MAXN], N;  
void DFS (int r){  
    for(int i=1;i<=N;i++) U[i]=0;  
    empty_stack();  
    push(r); U[r]=1; P[r]=0;  
    while(not_empty()){ int t=look();  
        if(G[t][0]>0){  
            int v=G[t][G[t][0]--];  
            if(!U[v]) {U[v]=1; P[v]=t; push(v);}  
        } else pop();  
    } }
```

# Сложност на DFS ?

Сложността на обхождане зависи от имплементацията:

|                              | <b>Матрица на съседство</b> | <b>Списък на наследниците</b> | <b>Списък на ребрата</b> |
|------------------------------|-----------------------------|-------------------------------|--------------------------|
| <b>Обхождане в дълбочина</b> | $\Theta(n^2)$               | $\Theta(n + m)$               | $\Theta(n * m)$          |

Източник: стр. 276 в книгата „Програмиране = ++Алгоритми;“ от Преслав Наков и Панайот Добриков.

В таблицата **n** е броят на върховете, а **m** броят на ребрата.

**Забележка**: „Списък на наследниците“ е еквивалентно за „Списък на съседите“, но е по-често използвано за ориентирани графи.

# Дефиниции

- ▶ В тази тема ще разглеждаме **графи с цени на ребрата**  
$$G(V, E), \ c:E \rightarrow \mathbb{R}^+$$
- ▶ Цената на реброто може да носи различни имена – **дължина, ширина, тегло** и т.н.. Без да се изменя същността на разглежданата задача
- ▶ Ще използваме най-популярното име за цената – **дължина** на реброто

# Дефиниции

- ▶ Нека  $\pi = v_0, v_1, \dots, v_L$  е път в графа от  $v_0$  до  $v_L$ .
- ▶ Дефинираме **дължина** на път

$$c(\pi) := c(v_0, v_1) + c(v_1, v_2) + \dots + c(v_{L-1}, v_L)$$

- ▶ Пътят  $\pi_0$  е най-къс път (НКП) в графа от  $v_0$  до  $v_L$ , ако за всеки друг път  $\pi$  от  $v_0$  до  $v_L$  е в сила

$$c(\pi_0) \leq c(\pi).$$

# Задачи

Даден е  $G(V, E)$ ,  $c:E \rightarrow \mathcal{R}^+$ :

- ▶ Да се намери НКП от зададен връх  $v$  до друг зададен връх  $w$  на графа;
- ▶ Да се намери НКП от зададен връх  $v$  до всеки друг връх на графа;
- ▶ Да се намери НКП от всеки връх на графа до всеки друг негов връх.

**Всички тези задачи са еквивалентни.**

# Дефиниции

Нека  $G(V, E)$ ,  $c:E\rightarrow\mathbb{R}^+$ , а  $T(V, E')$ ,  $E' \subseteq E$ , е дърво с корен  $v \in V$  такова, че всеки път от  $v$  до друг връх  $w$  в  $T$  е най-къс път в  $G$  от  $v$  до  $w$ , т.е. **дърво на най-къси пътища** (ДНКП) от  $v$  до всички останали върхове в  $G$ ;

Съществуването на ДНКП не е очевидно, но се доказва. Затова най-естествена от трите задачи е „от зададен връх  $v$  до всички останали“.

# Частният случай $c = \text{constant}$

**Задача.** Даден е граф  $G$  и връх  $r$ . Да се намерят най-къси пътища от  $r$  до всеки друг връх  $w$  в  $G$ .

**Теорема.** Нека  $G(V, E)$  е граф и  $c: E \rightarrow \mathbb{R}^+$  е такава, че  $c(e) = 1, \forall e \in E$ . Нека  $T(V, E)$  е ПД на  $G$  „в ширина“ с начален връх  $r$ . Тогава  $T$  е дърво на най-къси пътища в  $G$  от  $r$  до всички останали върхове.

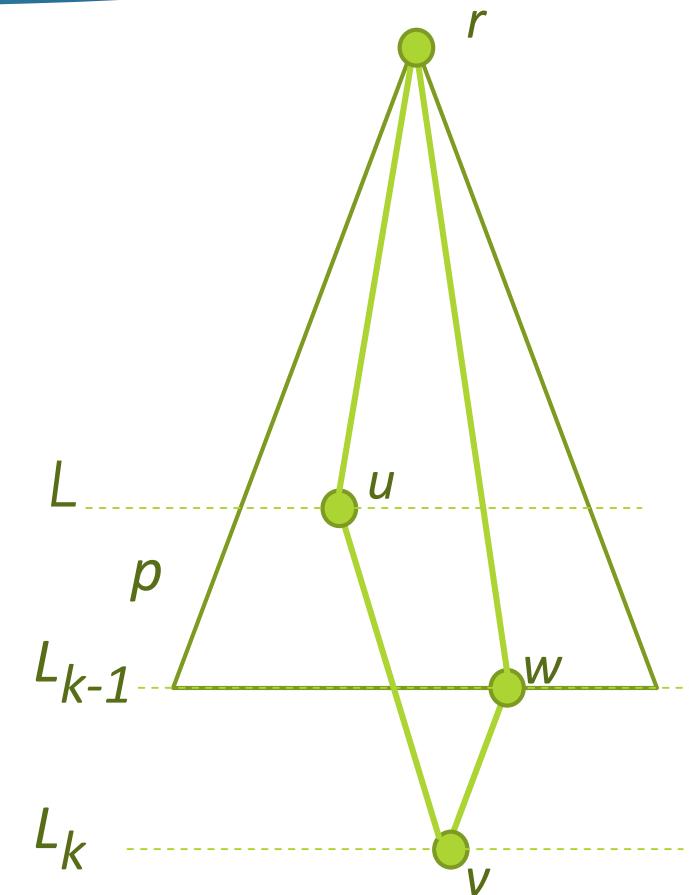
# Частният случай $c = \text{const}$

**Лема.** Дължината на НКП от  $r$  до всеки връх от  $L_i$  е  $i$ .

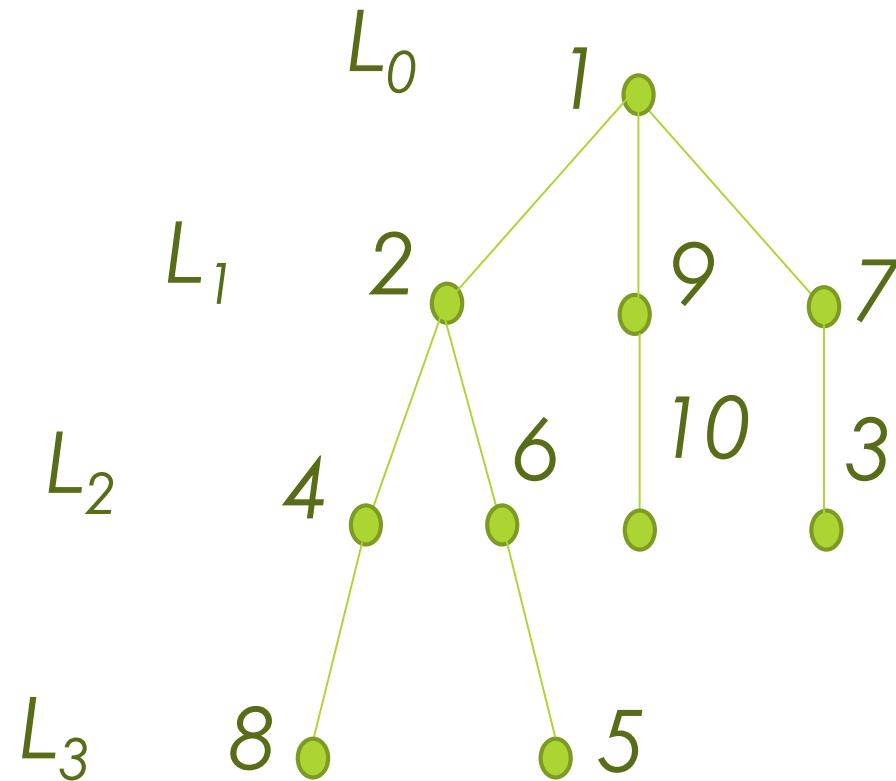
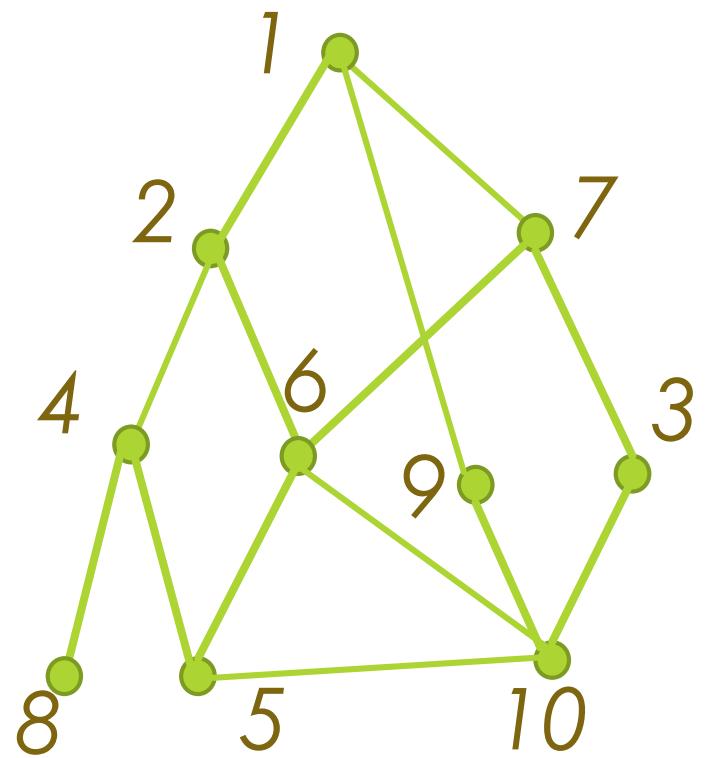
**Лема.** Очевидно за  $i=0$  тъй като  $L_0=\{r\}$  а  $c(v)=0$ . Нека лемата е всила за  $L_i$ ,  $i < k$ . Доп. че  $r, \dots, w, v$  не е НКП.

Нека  $r, \dots, u, v$  е НКП,  $p < k-1$ . Противоречие!

Защо  $v \notin L_{p+1}$ ,  $p+1 < k$ ?



# Частният случай $c = \text{const}$ ПД „в ширина“ е ДНКП



# Задача

**Като използвате функцията за построяване на покриващо дърво в ширина, напишете програма, която да намира най-къс път между два зададени върха на граф.**

**Графът е зададен по обичайния начин:**

N M

списък от M ребра

# Алгоритъм на Дейкстра

Нека  $G(V, E)$ ,  $c:E \rightarrow \mathbb{R}^+$ ,  $c \neq \text{const}$ ,  $r \in V$ . **Алгоритъмът на Дейкстра** строи ДНКП от  $r$  до всички останали върхове на  $G$ .

Нека  $V=\{1, 2, \dots, n\}$ , а за примера  $r = 1$ .

Ще смятаме че  $c()$  е дефинирана за всяко  $(i, j)$ , като ако  $(i, j)$  не е ребро,  $c(i, j)=\infty$  (много голямо число).

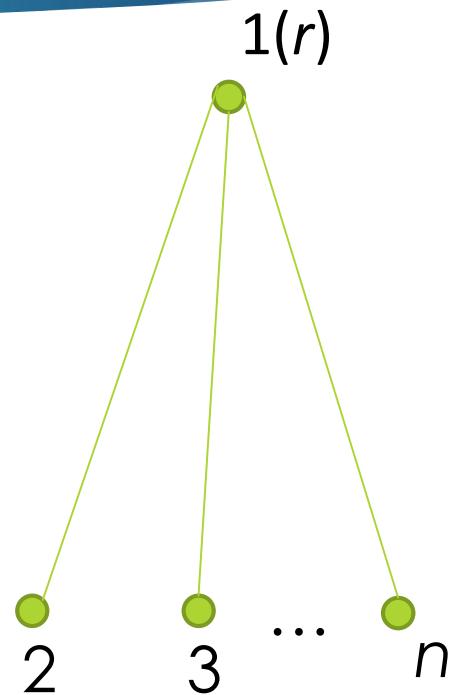
# Алгоритъм на Дейкстра – необходими структури

- ▶ Нека  $G$  е представен със списъци на съседите в `int G[MAXN][MAXN];`
- ▶ Цените на ребрата са в паралелен масив `int C[MAXN][MAXN];`
- ▶ `int d[MAXN]`,  $d[i]$  съдържа дължината на намерения да момента НКП от  $0$  до  $i=1,2,\dots,n$ ;
- ▶ `int p[MAXN]`,  $p[i]$  е бащата на  $i$ ,  $i=1,2,\dots,n$ , в дървото на НКП;
- ▶ `int U[MAXN]` - за отбелязване на обработените върхове, както в BFS/DFS;

# Алгоритъм на Дейкстра

ДНКП тук не се строи стъпка по стъпка, както в случая  $c=const$ , а се започва с едно грубо дърво на „временно“ НКП и това дърво на всяка стъпка се подобрява.

Началното дърво е показано на фигурата.



# Алгоритъм на Дейкстра

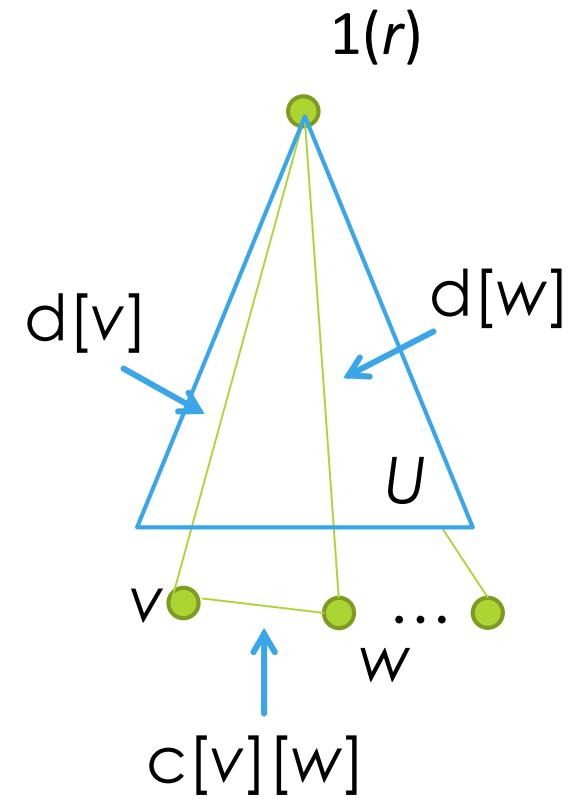
Релаксация на връх  $w$ ,  $w \notin U$  по връх  $v$  такъв, че  
 $d[v] \in \min, v \notin U$

Ако  $d[w] > d[v] + c[v][w]$

тогава

$$d[w] = d[v] + c[v][w]$$

и  $w$  се премества под  $v$  в ДНКП.



# Алгоритъм на Дейкстра – имплементация на C\С++

**Имплементацията на Дейкстра ще видим на Упражненията.**

# Алгоритъм на Дейкстра - извикване

```
#define INF 1000000000

int G[MAX][MAX], c[MAX][MAX], U[MAX],
p[MAX], d[MAX], n, m, r;
int main()
{ int u,v,w,i,j;
scanf("%d %d %d",&n,&m,&r);
for(i=1; i<=n; i++) {
    G[i][0]=0;
    for(j=1; j<=n; j++) c[i][j]=INF;
    if(i==j) c[i][j]=0;
}
...
for(i=1; i<=m; i++) {
    scanf("%d %d %d",&u,&v,&w);
    G[u][++G[u][0]]=v; G[v][++G[v][0]]=u;
    c[u][v]=c[v][u]=w;
}
dijkstra(r);
for(i=1; i<=n; i++)
printf("%d %d\n",i,p[i]);
return 0;
}
```

# Задачи

1. Напишете функция `dijkstra()`, която имплементира алгоритъма на Дейкстра.
2. Напишете главна програма, която въвежда граф с цени на ребрата, зададен с `N`, `M` и `M` ребра във вида:  
$$v \quad w$$
$$c(v,w) \quad \text{и извиква } \text{dijkstra}().$$
3. Допълнете главната програма с възможност да намира пътя между два зададени върха.

# Алгоритъм на Флойд-Уоршал

Имаме ориентиран граф с цени. За разлика от Дейкстра алгоритъмът на Флойд-Уоршал търси най-къси пътища от всеки връх до всеки връх.

Основна идея: Имаме ориентиран  $G(V, E, c)$ ,  $V = \{1, 2, 3, \dots, n\}$ .

Търсим най-къс път между върхове  $i$  и  $j$ , междинните върхове в този път търсим измежду върхове  $\{1, 2, \dots, k\}$ . Имаме два случая:

- 1)  $k$  участва в този най-къс път като междинен връх. Тогава търсим най-къс път между  $i$  и  $j$  с междинни върхове измежду  $\{1, 2, \dots, k-1\}$ ;
- 2)  $k$  не участва в този най-къс път. Тогава вече търсим два най-къси пътища, такъв между  $i$  и  $k$ , и такъв между  $k$  и  $j$ , и двата пътища с междинни върхове измежду  $\{1, \dots, k-1\}$ .

# Алгоритъм на Флойд – извикване на НКП от връх до друг връх

```
int G[MAXN][MAXN], N;  
void path(int v, int w)  
{ printf("%d ", v);  
  if(P[v][w] != 0)  
  { path(v, P[v][w]);  
    path(P[v][w], w);  
  }  
}
```

**Да допуснем, че сме използвали алгоритъма на Флойд, за да намерим  $P[v][w]$ . Проследете какво прави функцията `path()`.**

**Така пазим върхът между два върха**

# Алгоритъм на Флойд – подготовка

```
#define INF 2000000  
int main()  
{ int i,j,k;  
    for(i=1;i<=N;i++)  
    { for(j=1; j<=N ;j++)  
        {G[i][j]=INF; P[i][j]=0;}  
        G[i][i]=0;  
    }  
    въвеждане на цените  
    извикване на floyd()  
    въвеждане на два върха v,w  
    извеждане на НКП от v до w }
```

Имплементацията на  
алгоритъма ще видите на  
Упражнения

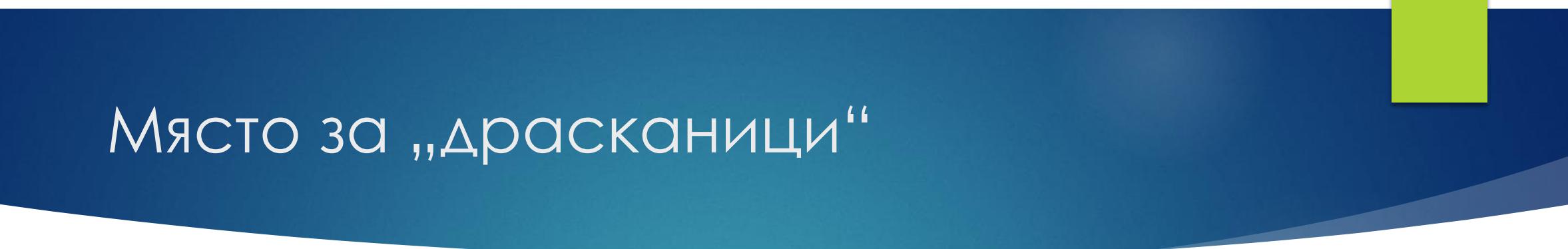
# Задачи

1. Довършете главната програма, която да намира НКП между два зададени върха;
2. Допълнете главната програма с възможност да намира най-дългия от всички НКП в графа – *диаметър на графа*.

# CSCB039 Алгоритми и програмиране

ЛЕКЦИЯ 10

гл. ас. д-р Слав Емилов Ангелов, НБУ  
проф. Красимир Манев



Място за „драсканици“

# Техниката „Разделяй и владей“

Нека е дадена задача  $\pi$  за съставяне на алгоритъм. Да означим с  $N$  размера на входните данни.

- ▶ **Разбиваме**  $\pi$  на  $p$  подзадачи от същия вид с размери  $N_1, N_2, \dots, N_p$ ;
- ▶ **Решаваме** всяка подзадача;
- ▶ **Сглобяваме** от решенията на подзадачите решение на задачата

# Пример

**Задача.** Да се пресметне биномният коефициент  $\binom{n}{k}$ .

**Решение:** „Разделяй и владей“

# Пример

**Задача.** Да се пресметне биномният коефициент  $\binom{n}{k}$ .

**Решение:** „Разделяй и владей“

Разбиване на подзадачи:

**Теорема на Паскал:**  $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

```
int binom(int n, int k)
{
    if(k==n || k==0) return 1;
    return binom(n-1,k) + binom(n-1,k-1);
}
```

# Пример

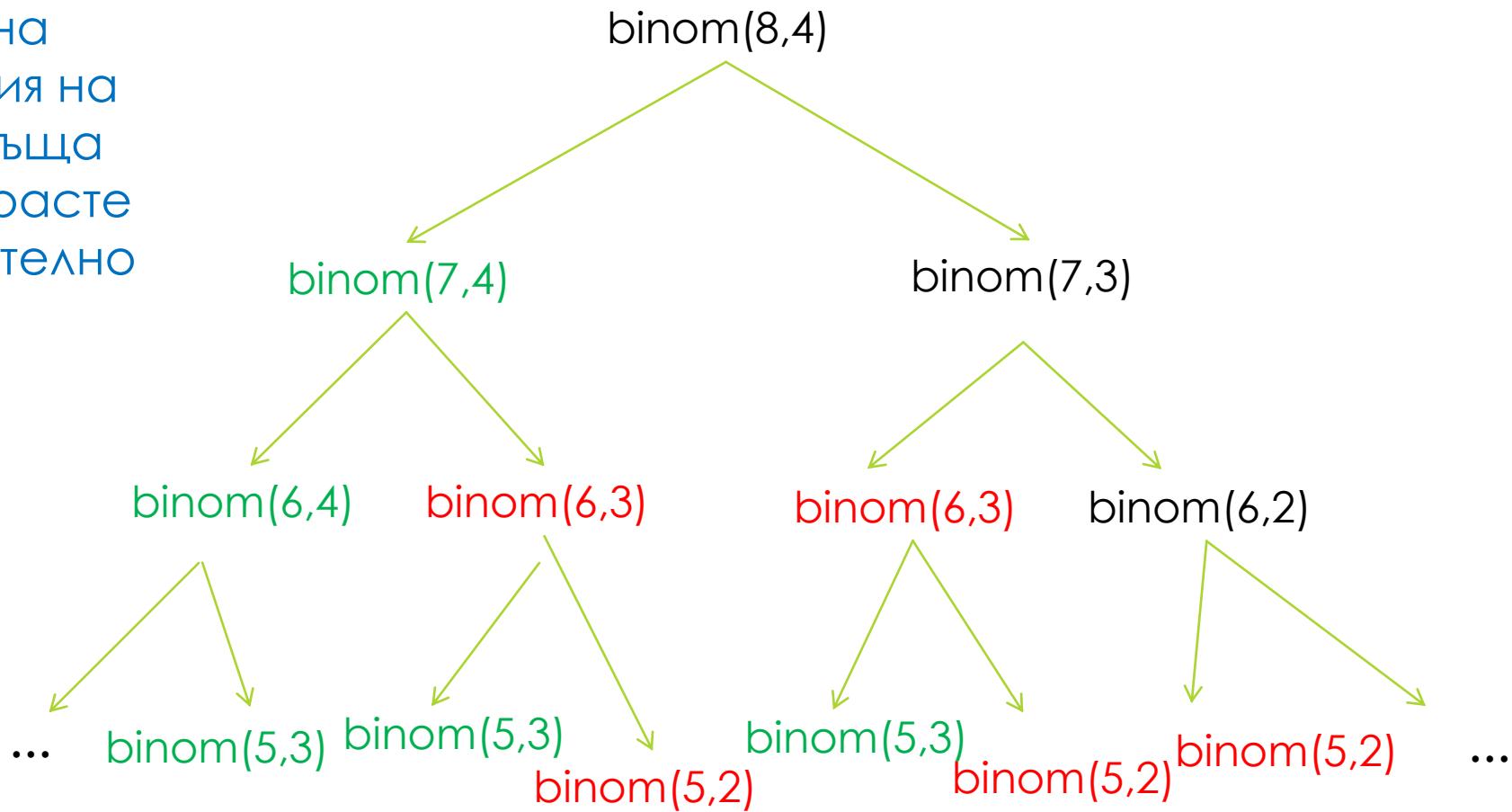
Сложност на този алгоритъм:

$$\begin{aligned}T(n) &= 2T(n-1) + C = \\&= 2(2T(n-2) + C) + C = 2^2T(n-2) + 3C = \\&= 2^2(2T(n-3) + C) + 3C = 2^3T(n-3) + 7C = \\&= \dots = \\&= 2^{n-1}T(1) + (2^{n-1}-1).C = O(2^n)\end{aligned}$$

Със сложност на  
рекурсивни функции  
ще се занимаем по-  
подробно следващата  
лекция

# Пример

Броят на  
изпълнения на  
една и съща  
функция расте  
застрашително



# Техниката „Разделяй и владей“

1. При прилагане на техниката „Разделяй и владей“ може да се получи многократно една и съща задача
2. Многократното решаване на една и съща задача увеличава сложността
3. Целта е да се избегне решаването по няколко пъти на една и съща задача

# Динамично програмиране

Нека е дадена задача  $\pi$  за решаване с алгоритъм. Да означим с  $N$  размера на входните данни.

- ▶ Разбиваме  $\pi$  на  $p$  подзадачи от същия вид с размери  $N_1, N_2, \dots, N_p$ ;
- ▶ Решаваме всяка подзадача, **която не сме решавали до момента и запомняме решението**, а за решаваните - ползваме запомненото решение;
- ▶ Сглобяваме от решенията на подзадачите решение на задачата.

# „Рекурсия с мемоизация“

**Задача.** Да се пресметне биномният коефициент  $\binom{n}{k}$ .

```
int b[MAXN][MAXN]={0};  
int binom(int n, int k)  
{ if(k==n||k==0) {b[n][k]=1; return 1;}  
    if(b[n-1][k]==0) b[n-1][k]=binom(n-1,k);  
    if(b[n-1][k-1]==0) b[n-1][k-1]=binom(n-1,k-1);  
    b[n][k]=b[n-1][k]+b[n-1][k-1];  
    return b[n][k];  
}
```

Да се намери  
сложността по  
време в този  
случай не е  
лесно!

# Итеративно решение

```
int b[MAXN][MAXN]={0};  
int binom(int n, int k)  
{ int i,j;  
    b[0][0]=b[1][0]=b[1][1]=1;  
    for(i=2;i<=n;i++)  
    { b[i][0]=b[i][i]=1;  
        for(j=1;j<i;j++)  
            b[i][j]=b[i-1][j]+b[i-1][j-1];  
    }  
    return b[n][k];  
}
```

**Сложность ?**

# Итеративно решение

Сложността на итеративното решение:

$$\begin{aligned}T(n) &= c \cdot 1 + c \cdot 2 + \dots + c \cdot (n+1) = \\&= c(n+1)(n+2)/2 = O(n^2)\end{aligned}$$

# Основен проблем на динамичното програмиране

Необходимост от **памет** за запомняне на решенията на подзадачите.

**Не за всяка задача, подзадачите се запомнят лесно – сортиране ??**

# Пример с „линейна“ таблица

**Задача  $\pi(N)$ .**  $N$  човека чакат на опашка за билети, като времето за което  $i$ -тият човек ще си купи билет е  $t_i$ ,  $i=1,2,\dots,N$ . Ако двама съседи в опашката,  $i$ -тият и  $(i+1)$ -ят, се кооперират, купуват билетите за време  $s_i$ ,  $i=1,2,\dots,N-1$ . Да се намери минималното време, за което може всички в опашката да си купят билети.

# Пример с „линейна“ таблица - решение

Разбиваме задачата  $\pi(N)$  на подзадачи  $\pi(i)$ ,  $i=1, 2, \dots, N$ , където  $\pi(i)$  е задачата: да се намери  $t_{\min}$  време за което първите  $i$  в опашката ще си купят билети.

**Решението на всяка от подзадачите е число** и се запомня в **„линейна“ таблица** (едномерен масив)  $M[1:N]$ , като в  $M[i]$  записваме решението на  $\pi(i)$ .

**Решението на подзадачата  $\pi(i)$  е решение и на задачата  $\pi(N)$ !**

# Принцип за оптималност

За да решим задачата, тряба да формулираме съответно твърдение – **принцип за оптималност.**

**Теорема.**

$$M[1] = t_1;$$

$$M[2] = \min\{t_1+t_2, s_1\},$$

...

$$M[i] = \min\{t_i + M[i-1], s_{i-1} + M[i-2]\}, \text{ за } i > 2.$$

Така преценяваме дали е по-изгодно да се кооперират или не

Зашо ?

# Пример с „линейна“ таблица

```
int t[MAXN], s[MAXN], M[MAXN];  
  
int opt(){    M[1]=t[1];  
    M[2]=t[1]+t[2];  
    if(M[2]>s[1]) M[2]=s[1];  
    for(int i=3;i<=N;i++)  
    {    M[i]=t[i]+M[i-1];  
        if(M[i]>s[i-1]+M[i-2])  
            M[i]=s[i-1]+M[i-2];  
    }  
    return M[N];  
}
```

**Кой обект е линейната таблица в момента ?**

# Слаба и силна форма на резултата с ДП

Практически всяка задача, която се решава с ДП може да се постави в две форми:

- **слаба**, при която се търси някаква „оптимална“ стойност;
- **силна**, при която се търси някаква структура в данните, при която се получава оптималното решение.

Силната форма на разглежданата задача е да се поиска да се посочи кооперирането, при което се получава оптимумът.

# Пример с „линейна“ таблица

```
int t[MAXN], s[MAXN], M[MAXN];  
  
void restore(int i){ if(i==1) {printf("(1)"); return; }  
    if(i==2)  
    { if(M[2]== t[1]+t[2])  
        { printf("(1) (2)"); return; }  
        else { printf("(1 2)"); return; };  
    }  
    if(M[i]==t[i]+M[i-1]) { restore(i-1); printf("(%d)", i); }  
    else  
    { restore(i-2); printf("(%d %d)", i-1, i); }  
}
```

**Какво прави тази функция ?**

# Пример с „линейна“ таблица

|   | 1 | 2 | 3  | 4  | 5 | 6 | 7 | 8 |
|---|---|---|----|----|---|---|---|---|
| t | 5 | 2 | 7  | 8  | 3 | 9 | 4 | 2 |
| s | 6 | 3 | 10 | 12 | 8 | 8 | 6 |   |

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7  | 8   |
|---|---|---|---|---|---|---|----|-----|
| M | 5 | 1 | 6 | 2 | 8 | 2 | 16 | 1,2 |
|   |   |   |   |   |   |   |    |     |

Какво е това ?

Решение на **силната задача**:

(1,2) (3,4) (5) (6,7) (8) или (1) (2,3) (4) (5) (6,7) (8).

# Пример с „линейна“ таблица

|   | 1 | 2 | 3  | 4  | 5 | 6 | 7 | 8 |
|---|---|---|----|----|---|---|---|---|
| t | 5 | 2 | 7  | 8  | 3 | 9 | 4 | 2 |
| s | 6 | 3 | 10 | 12 | 8 | 8 | 6 |   |

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7  | 8   |
|---|---|---|---|---|---|---|----|-----|
| M | 5 | 1 | 6 | 2 | 8 | 2 | 16 | 1,2 |
|   |   |   |   |   |   |   |    |     |

Решение на **силната задача**:

(1,2) (3,4) (5) (6,7) (8) или (1) (2,3) (4) (5) (6,7) (8).

Какво е това ?  
Избора ни на всяка стъпка към крайното решение.

# Пример с „триъгълна“ таблица

**Задача  $\pi(N)$ .**  $N$  човека чакат на опашка за билети, като времето за което  $i$ -тият човек ще си купи билет е  $t_{1,i}$ ,  $i=1, 2, \dots, N$ . Ако двама съседи в опашката,  $i$ -тият и  $(i+1)$ -ят се кооперират, купуват билетите за време  $t_{2,i}$ ,  $i=1, 2, \dots, N-1$ .

Ако  $j$  съседи в опашката - от  $i$ -тия до  $(i+j-1)$ -вия се кооперират, купуват билетите за време  $t_{j,i}$ ,  $i=1, 2, \dots, N-j+1$ , и т.н.. А всички заедно биха си купили билети за време  $t_{N,1}$ .

**Да се намери минималното време, за което може всички в опашката да си купят билети.**

# Пример с „триъгълна“ таблица - решение

Разбиваме задачата  $\pi(N)$  на подзадачи  $\pi(j, i)$ ,  $j = 1, 2, \dots, N$ ;  
 $i=1,2,\dots,N-j+1$ , където  $\pi(j, i)$  е задачата да се намери  $\min$  време, за  
което ј последователни чакащи в опашката, започващи от  $i$ -тия ще си купят  
билети.

Тъй като решението на всяка от подзадачите е едно число, решенията се  
запомнят лесно в **„триъгълна“ таблица** (половинката на двумерен масив)  
 $M[1:N][1:N]$ , като в  $M[j][i]$  записваме решението на  $\pi(j, i)$

**Решението на подзадачата  $\pi(N, 1)$  е решение и на задачата  $\pi(N)$ .**

# Пример с „триъгълна“ таблица

**Теорема.** а)  $M[1][i] = t_{1,i}$  за  $i=1,2,\dots,N$ .

б)  $M[j][i] = \min\{ M_{1,i} + M_{j-1,i+1}, M_{2,i} + M_{j-2,i+2}, \dots, M_{k,i} + M_{j-k,i+k}, \dots, M_{j-1,i} + M_{1,i+j-1}, t_{j,i} \}$

за  $j=2,3,\dots,N$ .

|     | 1         |  | i                      |              |  | ...          | N                        |
|-----|-----------|--|------------------------|--------------|--|--------------|--------------------------|
| 1   | $t_{1,1}$ |  | <sup>1</sup> $t_{1,i}$ |              |  |              | $t_{1,N}$ <sup>j-1</sup> |
| 2   |           |  | <sup>2</sup>           |              |  | ...          |                          |
| 3   |           |  | ...                    |              |  | <sup>2</sup> |                          |
| ... |           |  | <sup>j-1</sup>         | <sup>1</sup> |  |              |                          |
| j   |           |  |                        |              |  |              |                          |
| ... |           |  |                        |              |  |              |                          |
| N   |           |  |                        |              |  |              |                          |

# Примерът с „триъгълна“ таблица

**Задача.** Напишете програма, която решава поставената задача.

**Допълнително.** Използвайки образеца от предната задача, допълнете решението с възможност **да показва как се групират чакащите** при купуване на билетите, за да се получи минималното време.

# „Правоъгълна“ таблица

**Зад.** Дадени са редица  $\alpha = a_1, a_2, \dots, a_M$  и редица  $\beta = b_1, b_2, \dots, b_N$ . Да се намери дължината на **най-дългата обща подредица** на  $\alpha$  и  $\beta$ .

**Общата подредица на две редици** трябва да съдържа елементи които ги има и в двете редици, да са в същия ред, както в редиците без да е необходимо да са последователни. Например общи подредици на aaabddc и abcdd са a, ab, abc, abdd и др.

# „Правоъгълна“ таблица - решение

Разбиваме задачата  $\pi(N)$  на подзадачи  $\pi(i, j)$ ,  $i=0, 1, 2, \dots, M$ ;  
 $j=0, 1, \dots, N$ , където  $\pi(i, j)$  е задачата да се намери дължината на най-  
дългата обща подредица на  $\alpha_i = a_1, \dots, a_i$  и  $\beta_j = b_1, \dots, b_j$ ,  $\alpha_0 = \beta_0 = \varepsilon$

Тъй като решението на всяка от подзадачите е едно число, запомня се  
лесно в „правоъгълна“ таблица  $L[1:M][1:N]$ , като в  $L[i][j]$   
записваме решението на  $\pi(i, j)$ .

**Решението на подзадачата  $\pi(M, N)$  е решение и на задачата  $\pi(N)$ .**

# „Правоъгълна“ таблица

**Теорема.**

$$L[i][0] = 0, \quad i = 0, 1, 2, \dots, M$$

$$L[0][j] = 0, \quad j = 0, 1, 2, \dots, N$$

$$L[i][j] = \begin{cases} L[i - 1][j - 1] + 1, & a_i = b_j \\ \max\{L[i - 1][j], L[i][j - 1]\}, & a_i \neq b_j \end{cases}$$

# „Правоъгълна“ таблица

| <b>L</b> |   | <b>a</b> | <b>a</b> | <b>a</b> | <b>b</b> | <b>d</b> | <b>d</b> | <b>c</b> |
|----------|---|----------|----------|----------|----------|----------|----------|----------|
|          | 0 | 0        | 0        | 0        | 0        | 0        | 0        | 0        |
| <b>a</b> | 0 | 1        | 1        | 1        | 1        | 1        | 1        | 1        |
| <b>b</b> | 0 | 1        | 1        | 1        | 2        | 2        | 2        | 2        |
| <b>c</b> | 0 | 1        | 1        | 1        | 2        | 2        | 2        | 3        |
| <b>d</b> | 0 | 1        | 1        | 1        | 2        | 3        | 3        | 3        |
| <b>d</b> | 0 | 1        | 1        | 1        | 2        | 3        | 4        | <b>4</b> |

# „Правоъгълна“ таблица

Ако трябва да решим силната форма, при която се търси и самата подредица, вървим назад по пътя, по който е изчислен максимумът и събираме елементите, които са еднакви в двете редици.

|   |   | a | a | a | b | d | d | c |
|---|---|---|---|---|---|---|---|---|
|   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| b | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 |
| c | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 |
| d | 0 | 1 | 1 | 1 | 2 | 3 | 3 | 3 |
| d | 0 | 1 | 1 | 1 | 2 | 3 | 4 | 4 |

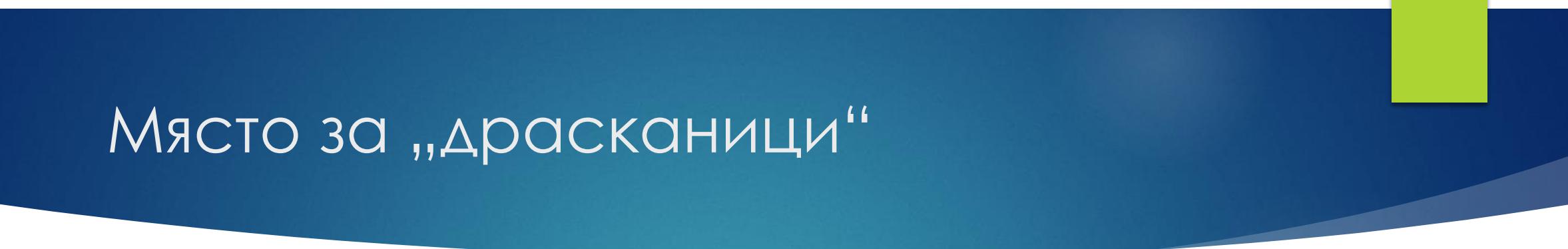
# Задача

**Каква е сложността по време и памет за всяка от задачите, които са решени в лекцията с техниката ДП?**

# CSCB039 Алгоритми и програмиране

ЛЕКЦИЯ 11

гл. ас. д-р Слав Емилов Ангелов, НБУ  
проф. Красимир Манев



Място за „драсканици“

# Рекурсивни функции

**Рекурсивна** е такава функция, в тялото на която се среща извикване (или няколко извиквания) на същата функция, **с размер на входните данни по-малък отколкото в извикването на същата функция**. Например,

```
int recf(int n) {  
    ...  
    int x = recf(n1);  
    ...  
}  
, където  $n1 < n$ .
```

# Сложност на рекурсивни функции

При оценяване на **сложността на рекурсивни** функции възниква следният проблем – в оценката непременно участва сложността на рекурсивното извикване, която всъщност търсим. Получава се израз от вида:

$$T_{\text{recf}}(n) = T_{\text{recf}}(n_1) + f(n)$$

или (при няколко извиквания):

$$T_{\text{recf}}(n)=T_{\text{recf}}(n_1)+T_{\text{recf}}(n_2)+\dots+T_{\text{recf}}(n_k)+f(n).$$



Това е сложността на всичко останало  
(без самото рекурсивно извикване)

# Сложност на рекурсивни функции

Тъй като всяка съмислена рекурсивна функция има „дъно“, т.е. стойност на аргумента  $a$ , при която не се налага вече рекурсивно извикване, лесно може да се пресметне  $T(a)$  – константа  $c$ .

Така, сложността се описва с

$$T_{\text{recf}}(a) = c; \quad T_{\text{recf}}(n) = T_{\text{recf}}(n_1) + f(n)$$

или

$$T_{\text{recf}}(a) = c; \quad T_{\text{recf}}(n) = T_{\text{recf}}(n_1) + T_{\text{recf}}(n_2) + \dots + T_{\text{recf}}(n_k) + f(n).$$

Наричано **рекурентно отношение (РО)**.

# Пример 1

Познатата функция  $n!$  (факториел) се дефинира с РО:

$$0! = 1; \quad n! = n(n-1)!, \quad n > 0$$

и се пресмята функцията

```
int fac(n) {  
    if(n==0 || n==1) return 1;  
    return n*fac(n-1);  
}
```

затова  $T_{\text{fac}}(n)$  се описва от РО:

$$T_{\text{fac}}(0) = T_{\text{fac}}(1) = \text{const};$$

$$T_{\text{fac}}(n) = T_{\text{fac}}(n-1)+1, \quad n > 0.$$

## Пример 2

Редицата на Фибоначи  $F_0, F_1, \dots, F_n, \dots$  е дефинирана с РО:

$$F_0 = F_1 = 1; \quad F_n = F_{n-1} + F_{n-2}$$

и се пресмята функцията:

```
int Fib(n) {  
    if(n==0 || n==1) return 1;  
    return Fib(n-1)+Fib(n-2);  
}
```

Затова  $T_{\text{Fib}}(n)$  се описва от РО:

$$T_{\text{Fib}}(0) = T_{\text{Fib}}(1) = \text{const};$$

$$T_{\text{Fib}}(n) = T_{\text{Fib}}(n-1) + T_{\text{Fib}}(n-2) + 1, \quad n > 1.$$

# Пример 3

Биномният коефициент  $B(n, m)$  можем да пресметнем, съгласно теоремата на Паскал, с рекурсивната функция:

```
int bin(n,k) {  
    if(k==0) return 1;  
    if(k==1) return n;  
    return b(n-1,k) + b(n-1,k-1);  
}
```

затова  $T_{bin}(n)$  се описва от РО:

$$T_{bin}(0) = T_{bin}(1) = \text{const};$$

$$T_{Fib}(n) = 2*T_{Fib}(n-1) + 1, \quad n > 1$$

# Още примери

## Summary: Recurrence Relations of Recursive Algorithms



| Recursive Algorithms             | Number of Subproblems | Additional Operations           | Time Complexity Recurrence        |
|----------------------------------|-----------------------|---------------------------------|-----------------------------------|
| Reversing an array               | 1                     | Swapping                        | $T(n) = T(n-2) + c$               |
| Binary Search                    | 1                     | Comparison                      | $T(n) = T(n/2) + c$               |
| Merge Sort                       | 2                     | Merging                         | $T(n) = 2T(n/2) + cn$             |
| Quick Sort                       | 2                     | Partition                       | $T(n) = T(i) + T(n - i - 1) + cn$ |
| Karatsuba Algorithm              | 3                     | Bitwise addition and Shifting   | $T(n) = 3T(n/2) + cn$             |
| Strassen's matrix multiplication | 7                     | Matrix addition and subtraction | $T(n) = 7T(n/2) + cn^2$           |
| Finding nth Fibonacci            | 2                     | Addition                        | $T(n) = T(n-1) + T(n-2) + c$      |

# Алгоритъм на Каратсуба

Умножава две големи числа с  $n$  цифри. Сложността му е  $O(\log 3) = O(n^{1.58})$ .

Пример: Искаме да умножим числата 12 345 и 6789. Общата база е 1000.

$$12\ 345 = 12 \cdot 1000 + 345, \quad 6789 = 6 \cdot 1000 + 789.$$

$$Z_2 = 12 \cdot 6 = 72,$$

$$Z_0 = 345 \cdot 789 = 272\ 205,$$

$$Z_1 = (12+345) \cdot (6+789) - Z_2 - Z_0 = 11\ 538.$$

$$\text{Резултат} = Z_2 \cdot \text{база} \cdot \text{база} + Z_1 \cdot \text{база} + Z_0 = 72 \cdot 1000 \cdot 1000 + 11\ 538 \cdot 100 + 272\ 205 = 83\ 810\ 205.$$

Тук се вижда рекурсията

Каратсуба се използва, когато.

# Алгоритми за умножение на числа

- ▶ **Алгоритъм на Каратсуба** – за числа > 320-640 бита. Сложност  $O(n^{1.58})$ .
- ▶ **Toom–Cook algorithm** (1963) – модификация на Каратсуба със сложност  $O(n^{\frac{\log 5}{\log 3}}) = O(n^{1.46})$ . Използва се за средно големи числа (10 000 – 40 000 цифри).
- ▶ **Schönhage–Strassen algorithm** (1971) – използва се за много големи числа (33 000 – 150 000 цифри). Сложност  $O(n * \log n * \log \log n)$ .
- ▶ **Fürer's algorithm** (2007) – по-бърз от предходния за числа с поне  $2^{2^{64}}$  цифри...
- ▶ **Модификация на Fürer's algorithm (2019)** - Harvey и van der Hoeven публикуват първият алгоритъм за умножение на цели числа с асимптотична сложност  $O(n * \log n)$ . Това е предполагаемият теоретичен лимит. Harvey, David; Van Der Hoeven, Joris (2019-04-12). "Integer multiplication in time  $O(n \log n)$ ". Annals of Mathematics.

# Умножение на числа с плаваща запетая

**Изброените алгоритми акцентират върху цели числа. А какво става за числа с плаваща запетая?**

# Решаване на РО

**Решаването на РО не е възможно в общия случай.** Съществуват техники за решаване на някои най-често срещани РО.

Ще се запознаем с „*техниката с итериране*“, която е достатъчна за уводния курс Алгоритми.

# Решение за Пример 1

За да решим РО от Пример 1, итерираме общия случай, т.е. разписваме равенството, докато стигнем до крайния случай:

$$T_{\text{fac}}(n) = T_{\text{fac}}(n-1) + 1$$

$$T_{\text{fac}}(n-1) = T_{\text{fac}}(n-2) + 1$$

...

$$T_{\text{fac}}(2) = T_{\text{fac}}(1) + 1$$

$$T_{\text{fac}}(1) = \text{const}$$

и събираме така получените равенства. Свързаните със стрелки членове се унищожават и получаваме:

$$T_{\text{fac}}(n) = 1 + 1 + \dots + 1 + \text{const} = n - 1 + \text{const} = O(n).$$

# Решение за Пример 3

За РО от Пример 3 не е достатъчно разписване на равенството, защото неизвестното в дясно е с коефициент 2. Затова, за да елиминираме ненужните членове, умножаваме последователно със степени на 2:

$$T_{\text{bin}}(n) = 2 \cdot T_{\text{bin}}(n-1) + 1 \quad \times 2^0=1$$

$$T_{\text{bin}}(n-1) = 2 \cdot T_{\text{bin}}(n-2) + 1 \quad \times 2^1$$

...

$$T_{\text{bin}}(2) = 2 \cdot T_{\text{bin}}(1) + 1 \quad \times 2^{n-2}$$

$$T_{\text{bin}}(1) = \text{const;} \quad \times 2^{n-1}$$

И получаваме

# Решение за Пример 3

$$T_{\text{bin}}(n) = 2^1 \cdot T_{\text{bin}}(n-1) + 2^0$$

$$2^1 \cdot T_{\text{bin}}(n-1) = 2^2 \cdot T_{\text{bin}}(n-2) + 2^1$$

$$2^2 \cdot T_{\text{bin}}(n-2) = 2^3 \cdot T_{\text{bin}}(n-3) + 2^2$$

...

$$2^{n-2} \cdot T_{\text{bin}}(2) = 2^{n-1} \cdot T_{\text{bin}}(1) + 2^{n-2}$$

$$2^{n-1} \cdot T_{\text{bin}}(1) = 2^{n-1} \cdot c$$

И след като съберем и направим приведението получаваме:

$$T_{\text{bin}}(n) = 2^0 + 2^1 + 2^2 + \dots + 2^{n-2} + 2^{n-1} + c2^{n-1} = 2^n - 1 + c2^{n-1} = O(2^n).$$

# НЯКОИ ВАЖНИ СУМИ

Както се вижда, при итериране трябва да може да се оцени някаква сума. Ето някои такива суми:

$$1 + 2 + \dots + n = n(n+1)/2 = O(n^2)$$

$$1^2 + 2^2 + \dots + n^2 = n(n+1)(2n+1)/6 = O(n^3)$$

$$2^0 + 2^1 + \dots + 2^n = 2(2^n - 1) = O(2^n)$$

$$3^0 + 3^1 + \dots + 3^n = (3(3^n - 1))/2 = O(3^n)$$

$$\log 1 + \log 2 + \dots + \log n = \log n! = O(n \cdot \log n)$$

# Решение за Пример 2

За РО от Пример 2 итерирането не е добра техника, заради двете неизвестни в дясното. Затова решаваме по друг начин. Изписваме общия случай така  $T(n) - T(n-1) - T(n-2) = 0$  и съставяме съответното уравнение

$x^2 - x - 1 = 0$  (с коефициентите от РО), по-големият корен на което е  $\alpha \approx 1,67$ . Решението в такъв случай е, за някаква константа  $a$ ,  $T(n) = O(a \cdot \alpha^n) = O(\alpha^n)$ .

# Рекурсия с мемоизация

При този вид рекурсия запомняме решенията така, че да избегнем повторения.  
**Това води до линеен брой извиквания на рекурсията?**

Не всяка рекурсия може да се момоизира ефикасно – опитайте с Merge sort.

# Задачи

**Решете РО:**

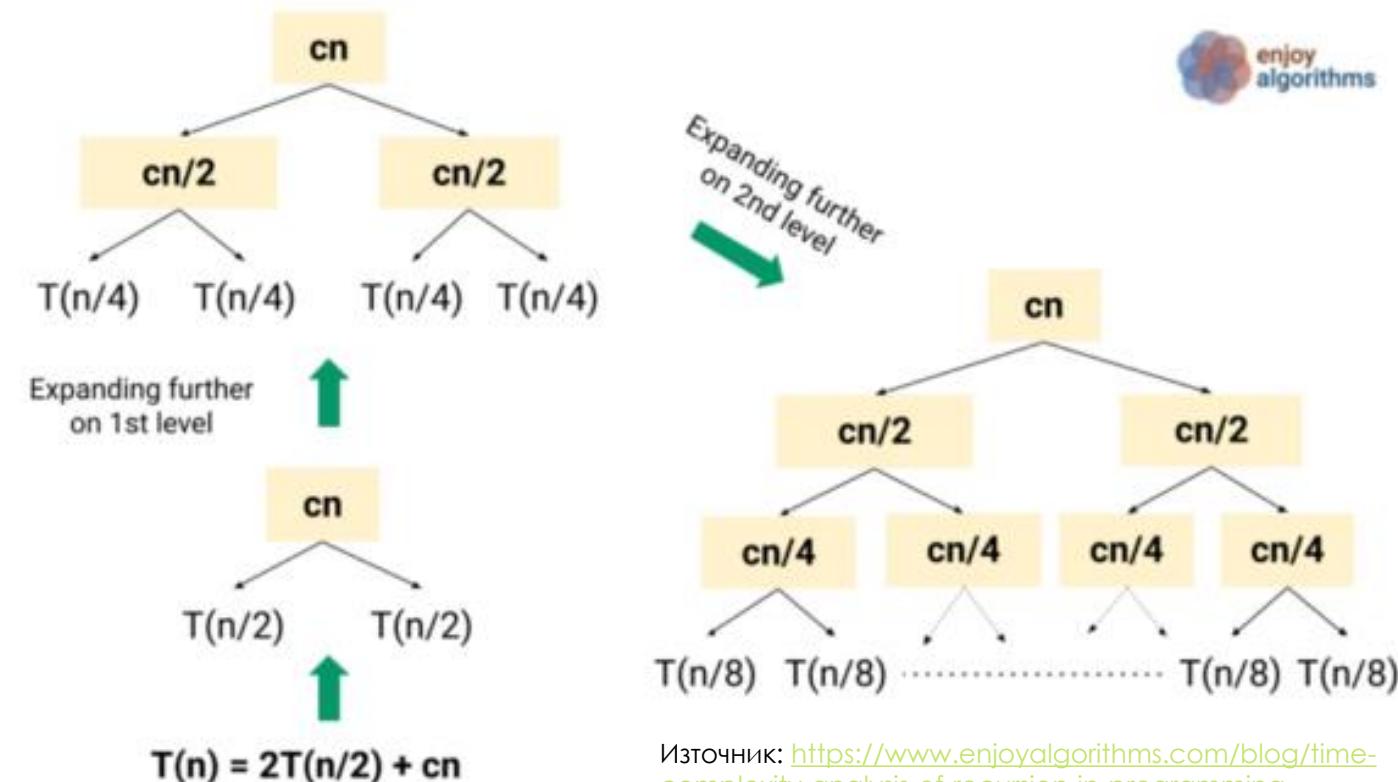
1.  $T(1)=\text{const}; T(n)=T(n-1)+n, n>1$
2.  $T(1)=\text{const}; T(n)=3 \cdot T(n-1)+1, n>1$
3.  $T(1)=\text{const}; T(n)=2 \cdot T(n-1)+n, n>1$
4.  $T(1)=\text{const}; T(n)=2 \cdot T(n-1)-T(n-2), n>1$

# Рекурсивни дървета

Можем да определим сложността на рекурсията на база на рекурсивни дървета:

Разписваме всички рекурсивни нива, а за всяко извикване записваме и сложността на допълнителните операции.

Така оценихме Merge sort преди няколко лекции.



Източник: <https://www.enjoyalgorithms.com/blog/time-complexity-analysis-of-recursion-in-programming>

# Мастър теорема

# Мастър теорема

Използва се, за да оценим рекурсия, която се базира на алгоритмичната схема „разделяй и владей“ при подразделяне на подзадачи с еднакъв размер.

Тоест тази теорема решава всички рекурсии от следния вид:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^k), \quad \text{където } a \geq 1, b > 1.$$

# Опростен вариант

Имаме задача с размер  $n$ , която се дели на  $a$  подзадачи. Всяка от подзадачите в с размер  $n/b$ , където  $a \geq 1$ ,  $b > 1$ . Всяка от  $a$ -те подзадачи се решава рекурсивно за време  $T\left(\frac{n}{b}\right)$ . Използва се  $O(n^k)$ , за да се покаже сложността на разделянето на подзадачи и тяхното сглобяване. Разграничават се три случая:

1.  $k < \log_b a \rightarrow T(n) = O(n^{\log_b a})$ ;
2.  $k = \log_b a \rightarrow T(n) = O(n^k \log n)$ ;
3.  $k > \log_b a \rightarrow T(n) = O(n^k)$ .

## Пример – двоично търсене

$$T(n) = T\left(\frac{n}{2}\right) + c \rightarrow a = 1, b = 2, k = 0 \text{ (защото } O(n^0) = const\text{).}$$

Понеже  $k = \log_2 1 = 0$ , то сме във втори случай на Мастър теоремата.  
Сложността на двоичното търсене е  $T(n)=O(\log n)$ .

## Пример 2 – матрично умножение по Страсен

$$T(n) = 7T\left(\frac{n}{2}\right) + cn^2 \rightarrow a = 7, b = 2, k = 2$$
$$\log_2 7 \cong 2.8 \rightarrow k < \log a.$$

Следователно сме в първи случай на Мастьр теоремата и сложността на матричното умножение по Страсен е  $T(n) = O(n^{\log_2 7}) \cong O(n^{2.8})$ .

# Пълен вариант

**Теорема:** (Мастър теорема)

За  $a \geq 1$ ,  $b > 1$  и  $f(n)$  - положителна е дадено рекурентното уравнение:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

*Случай 1:* Ако  $f(n) = O(n^{\log_b a - \varepsilon})$  за някое  $\varepsilon > 0$ , то  $T(n) = \Theta(n^{\log_b a})$

*Случай 2:* Ако  $f(n) = \Theta(n^{\log_b a})$ , то  $T(n) = \Theta(n^{\log_b a} \lg n)$

*Случай 3:* Ако  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  за някое  $\varepsilon > 0$  и

А дали не е  $\text{Log } n$  ?

$$\exists c \in (0, 1) \exists n_0 \in N \forall n \geq n_0: a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$$

, то  $T(n) = \Theta(f(n))$

Втората част от *Случай 3* се нарича *условие за регулярност*.

# Пример

$$T(n) = 4T(n/2) + n.$$

Решение:

$$a = 4, b = 2, \log_b a = 2.$$

$$f(n) = n = O(n^{2-0.1}) = O(n^{\log_2 4-0.1}).$$

Имаме първи случай, т.е  $T(n)=\Theta(n^2)$ .

## Пример 2

$$T(n) = 2T(n/2) + n.$$

Решение:

$$a = 4, \ b = 2, \ \log_b a = 1.$$

$f(n) = n = \Theta(n) = \Theta(n^{\log 2})$ . Следователно е втори случай на Мастьр теоремата и  $T(n) = \Theta(n \log n)$ .

## Пример 3

$$T(n) = 3T\left(\frac{n}{9}\right) + n \lg n$$

Имаме:

$$n \lg n = \Omega\left(n^{\frac{5}{6}}\right) = \Omega\left(n^{\log_9 3 + \frac{1}{3}}\right)$$

Освен това:

$$\forall n \geq 1: 3 \cdot \frac{n}{9} \cdot \lg \frac{n}{9} \leq \frac{1}{3} n \lg n$$

От *MTh 3*  $\Rightarrow T(n) = \Theta(n \lg n)$

## Пример 4

$$T(n) = 8T\left(\frac{n}{4}\right) + n \lg n$$

Имаме:

$$n \lg n = O\left(n^{\frac{3}{2}-0.1}\right) = O\left(n^{\log_4 8-0.1}\right) \stackrel{MTh\ 1}{\implies} T(n) = \Theta\left(n^{\frac{3}{2}}\right)$$

# Ограничения на Мастьр теоремата

Мастьр теоремата не може да бъде използвана, ако:

- ▶ **T(n)** не е полиномиална. Например  $T(n) = 2^n$ ;
- ▶ **T(n)** не е монотонна. Например  $T(n) = \sin n$ ;
- ▶ **a** не е константа. Например  $a=3n$ ;
- ▶ **a<1**.