ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

ТЕХНИКИ ЗА КРЕИРАЊЕ АЛГОРИТМИ 1

АЛГОРИТМИ И
ПОДАТОЧНИ СТРУКТУРИ
- предавања -

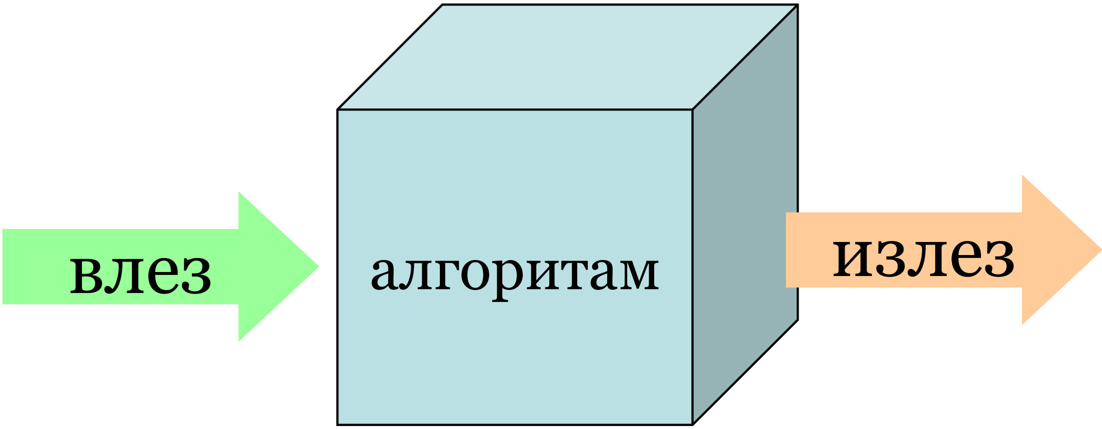
Содржина

- Вовед во пресметковни проблеми
- Начини на опишување на проблемите
- Псевдокод нотација
- Техники за дизајн на алгоритми
  - **Техника базирана на груба сила**
  - **Алчни алгоритми**
  - Раздели-и-владеј
  - Динамичко програмирање
  - Алгоритми со случајни броеви
  - Останати алгоритми

Што е тоа алгоритам?

- **Алгоритам** е секвенца од чекори/инструкции кои треба да се превземат со цел да се реши добро дефиниран *проблем*

- **Алгоритам** е методот за транслација на влезовите во соодветните излези



Пресметковни проблеми

- Првиот и најчесто најтешкиот чекор при решавањето на некој пресметковен проблем е - дефинирањето на суштината на проблемот.

- Секогаш треба да се имаат две основни прашања на ум:
  - “Дали алгоритамот работи правилно?”
  - “Колку време е потребно за неговото извршување?”

Пресметковни проблеми

- Еден проблем може да опишува цела класа пресметковни задачи.
- *Инстанца* на проблемот ќе биде еден конкретен влез за дадениот проблем.
- **ПРИМЕР:**
  Купувате млеко кое чини 51 денар со банкнота од 100 денари и треба да ви се врати кусур од 49 денари.
  - Проблем: Враќање кусур.
  - Инстанца на проблемот: Враќање кусур од 49 денари.

Како се задава еден алгоритам?

- Алгоритмите се состојат од чекори кои треба да се извршат
  - Описно задавање на чекорите во говорен јазик (лесно разбирливо за човекот, но не и за компјутерот)
  - Опишување преку псевдокод (недоволно разбирливо за компјутерот, но доволно апстрактно за да може да се преточи во програмски код)
  - Графички или визуелно прикажување на чекорите со помош на некој дијаграм (најчесто блок дијаграм)

Опишување преку псевдокод

- ***Псевдокодот*** е најчесто користениот начин за опишување на алгоритмите
  - Променливи, низи и аргументи
  - Специјална операција ***return*** го враќа резултатот со што завршува извршувањето на алгоритмот
  - Комбинирање на основните операции во мини-алгоритми наречени ***подалгоритми*** (*субрутини*)

Елементарни псевдокод команди

Доделување вредност

- Формат: $a \leftarrow b$
- Ефект: Променливата a ја прима вредноста b
- Пример: $b \leftarrow 2$ $\quad a \leftarrow b$
- Резултат: Вредноста на променливата a е 2

Елементарни псевдокод команди

Аритметички операции

- Формат: $a + b,\ a - b,\ a * b,\ a / b,\ a^b$
- Ефект: Собирање, одземање, множење, делење и степенување
- Пример:

```
DIST(x1, y1, x2, y2)
1  dx ← (x2 – x1)²
2  dy ← (y2 – y1)²
3  return sqrt(dx + dy)
```

- Резултат: DIST(x1, y1, x2, y2) го пресметува Евклидовото растојание помеѓу две точки.
  DIST(0,0,3,4) враќа 5

Елементарни псевдокод команди

- Условна проверка
  - Формат: **if** *A* is true B **else** C
  - Ефект: Ако е исполнет условот А тогаш се извршуваат инструкциите B, инаку се извршуваат инструкциите C
  - Пример: MAX(a, b)

```
MAX(a, b)
1   if a<b  return b
2   else  return a
```

  - Резултат: MAX(a, b) го враќа поголемиот од двата броеви a и b.
    MAX(1, 99) враќа 99

Елементарни псевдокод команди

- for циклуси
  - Формат: **for** $i \leftarrow a$ **to** b B
  - Ефект: Вредноста на *i* се поставува на *a* и се извршува B. Потоа *i* добива вредност *a+1* и пак се извршува B. Ова се повторува се додека *i* не добие вредност *b*.
  - Пример:

```
SUMINT(n)
1   sum ← 0
2   for i ← 1 to n   sum ← sum + i
3   return sum
```

  - Резултат: SUMINT(n) ја пресметува сумата на броевите од 1 до n. SUMINT(10) враќа 55

Елементарни псевдокод команди

- while циклуси
  - Формат: **while** *A* is true B
  - Ефект: Се проверува условот *A*. Ако е исполнет се извршува *B*. Се повторува проверката на исполнетост на *A* и ако е точно, тогаш се извршува пак *B* се додека *A* е неточно.
  - Пример:

```
ADDUNTIL(b)
1   i ← 1,  total ← i
2   while total ≤ b
3       i ← i + 1
4       total ← total + i
5   return i
```

  - Резултат: ADDUNTIL(b) го наоѓа најмалиот број i за кој важи дека сумата од 1 до тој број е поголема од b. ADDUNTIL(25) ќе врати 7

Елементарни псевдокод команди

пристап до елементи од низи

- Формат: $\mathbf{a_i}$
- Ефект: Го враќа *i*-тиот елемент од некоја низа дефинирана со **a** = $(a_1, \ldots, a_i, \ldots, a_n)$
- Пример:

```
FIBONACCI(n)
1    F1 ← 1,   F2 ← 1
2    for i ← 3 to n
3         Fi ← Fi-1 + Fi-2
4    return Fn
```

- Резултат: FIBONACCI(n) го пресметува n-тиот Фибоначиев број. FIBONACCI(8) враќа 21

Пример - рецепт

Аналогија помеѓу рецепт и алгоритам

1 влезни податоци
2. чекори на извршување
3. излез, резултат

3

1 ½ чаша сварена тиква
1 чаша кафеав шекер
1 мала лажичка сол
2 мали лажички цимет
1 мала лажичка ѓумбир
1 супена лажица меласа
3 јајца, малку изматени
1 ½ чаша хомогенизирано млеко
1 непечено тесто за пита

1

Измешајте ги тиквата, шекерот, солта, ѓумбирот, циметот и меласата. Додадете ги јајцата и млекото и добро измешајте. Ставете ја смесата во непеченото тесто за пита и печете во загреана рерна (220 °C) околу 40 до 45 минути или додека ножот [illegible] засекува питата остане чист.

2

Пример – рецепт, псевдокод

NAPRAVI_PITA_OD_TIKVA(*tikva*, *seker*, *sol*, *zacini*, *jajca*, *mleko*, *testo*)
ZAGREJ_RERNA(220)
polnenje ← ZAMESAJ(*tikva*, *seker*, *sol*, *zacini*, *jajca*, *mleko*)
pita ← SOSTAVI(*testo*, *polnenje*)
while noz za zasekuvanje ostane cist
    PECI(*pita*)

output “pitata e gotova”
return *pita*

*) NAPRAVI_PITA_OD_TIKVA го повикува подалгоритамот ZAMESAJ

Пример, псевдокод

ZAMESAJ(*tikva*, *seker*, *sol*, *zacini*, *jajca*, *mleko*)
1 *dlabokaCinija* ← zemi dlaboka cinija od plakarot
2 STAVI(*tikva*, *dlabokaCinija*)
3 STAVI(*seker*, *dlabokaCinija*)
4 STAVI(*sol*, *dlabokaCinija*)
5 STAVI(*zacini*, *dlabokaCinija*)
6 MESAJ(*dlabokaCinija*)
7 STAVI(*jajca*, *dlabokaCinija*)
8 STAVI(*mleko*, *dlabokaCinija*)
9 MESAJ(*dlabokaCinija*)
10 *polnenje* ← Sodrzina od *dlabokaCinija*
11 **return** *polnenje*

ЗА ДОМА:
Дајте графичка претстава на овој алгоритам!

Овој алгоритам не прави пита од тиква, туку ги **дефинира чекорите** кои треба да се извршат за да се направи питата.

За да се добие пита од тиква, треба да се конструира машина која ќе ги следи овие чекори за да се добие крајниот резултат!

Од псевдокод до имплементација

- **Псевдокод**: апстрактна секвенца од чекори која опишува решение на добро формулиран пресметковен проблем

- **Компјутерски код**: множество од детализирани инструкции кои ќе можат да се извршат на некој компјутер

Конверзијата на псевдокодот во компјутерски код не е едноставен процес!

Проблем на враќање кусур

- Треба да се врати кусур од 268 денари доколку на располагање ни се неограничен број банкноти и монети од 1000, 500, 100, 50, 10, 5, 2 и 1 денар. Исплатата треба да е со минимален број банкноти и монети!

Проблем на враќање кусур

Интуитивно решение
- најчест приод
- која е техниката?

```
VRATI_KUSUR(M)
Врати го целиот дел од делењето M/1000
Нека остаток биде остатокот што треба да му се врати на купувачот
Врати го целиот дел од делењето остаток/500
Нека остаток биде остатокот што треба да му се врати на купувачот
Врати го целиот дел од делењето остаток/100
Нека остаток биде остатокот што треба да му се врати на купувачот
Врати го целиот дел од делењето остаток/50
Нека остаток биде остатокот што треба да му се врати на купувачот
Врати го целиот дел од делењето остаток/10
Нека остаток биде остатокот што треба да му се врати на купувачот
Врати го целиот дел од делењето остаток/5
Нека остаток биде остатокот што треба да му се врати на купувачот
Врати го целиот дел од делењето остаток/2
Нека остаток биде остатокот што треба да му се врати на купувачот
Врати го целиот дел од делењето остаток/1
```

Што ќе врати овој алгоритам доколку треба да се врати кусур од 40 денари со монети од 25, 20, 10, 5 и 1 денар?

```
VRATI_KUSUR(M)
q ← M
r ← q/1000
q ← q − 1000·r
s ← q/500
q ← q − 500·s
t ← q/100
q ← q − 100·t
u ← q/50
q ← q − 50·u
v ← q/10
q ← q − 10·v
w ← q/5
q ← q − 5·w
x ← q/2
q ← q − 2·x
y ← q
return (r, s, t, u, v, w, x, y)
```

Истиот псевдокод но напишан на поблизок начин до имплементациската форма

Архетипи на алгорими

Архетипи – техники на решавање алгоритми

- техника на груба сила
- алчни (максималистички)
- раздели и владеј
- динамичко програмирање
- алгоритми со случајни броеви и
- алгоритми кои се враќаат наназад од резултатот

- алгоритми за дистрибуирано процесирање
- алгоритми кои работат со ограничувања
- други алгоритми...

Техника на груба сила (brute force)

- Наједноставна техника за решавање на проблеми
- Ги испитува сите можни случаи преку кои се доаѓа до резултатот
- Гарантира дека ќе најде резултат

Недостаток: Преголемо време и мемориски ресурси за извршување на ваквите алгоритми!

Техника на груба сила (brute force)

- Решение на проблемот враќање кусур со груба сила:

> Што ќе врати овој алгоритам доколку треба да се врати кусур од 40 денари со монети од 25, 20, 10, 5 и 1 денар?

BF_CHANGE(M, $\mathbf{c}$, d)

smallestNumberOfCoins $\leftarrow \infty$

for each $(i_1, \ldots, i_d)$ **from** $(0, \ldots, 0)$ **to** $(M/c_1, \ldots, M/c_d)$

    valueOfCoins $\leftarrow \sum_{k=1} i_k c_k$

    if *valueOfCoins* $= M$

        numberOfCoins $\leftarrow \sum_{k=1} i_k$

        if *numberOfCoins* $<$ *smallestNumberOfCoins*

            smallestNumberOfCoins $\leftarrow$ *numberOfCoins*

            bestChange $\leftarrow (i_1, i_2, \ldots, i_d)$

return(***bestChange***)

Техника на груба сила (brute force)

0	0	…	0	0
0	0	…	0	1
0	0	…	0	2
		…		
0	0	…	0	M/c_d
		…		
0	0	…	1	0
0	0	…	1	1
0	0	…	1	2
		…		
0	0	…	1	M/c_d
		…		
M/c_1	M/c_2	…	$M/c_{d-1} – 1$	0
M/c_1	M/c_2	…	$M/c_{d-1} – 1$	1
M/c_1	M/c_2	…	$M/c_{d-1} – 1$	2
		…		
M/c_1	M/c_2	…	$M/c_{d-1} – 1$	M/c_d
M/c_1	M/c_2	…	M/c_{d-1}	0
M/c_1	M/c_2	…	M/c_{d-1}	1
M/c_1	M/c_2	…	M/c_{d-1}	2
		…		
M/c_1	M/c_2	…	M/c_{d-1}	M/c_d

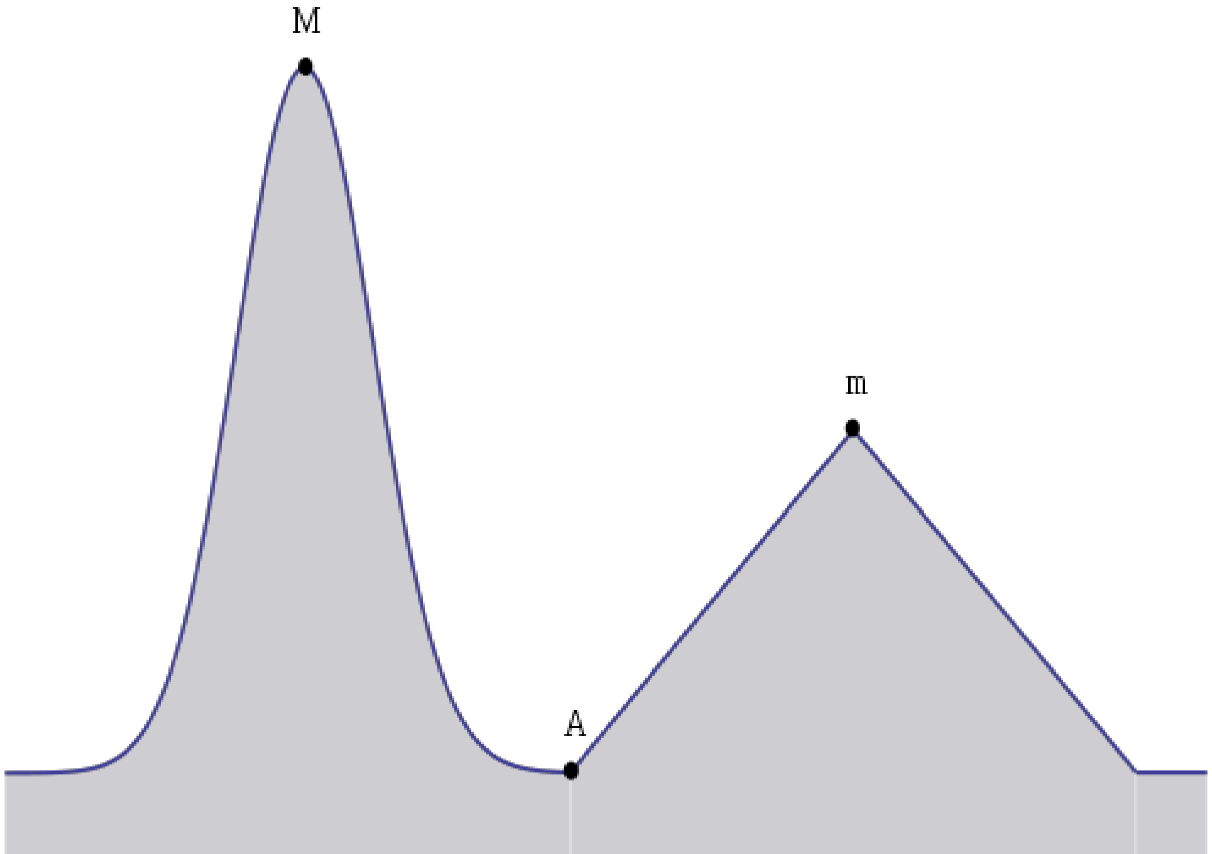
Оваа техника гарантира дека секогаш ќе се добива вистинското точно решение!

Приказ на сите можни комбинации што ќе се проверат во текот на извршувањето на алгоритмот, пред да се стаса до решението

Алчни (greedy) алгоритми

- секогаш го бараат **локалното оптимално** решение
- го избира најдоброто што е достапно во моментот
- најчесто креираат коректно решение но, само за дел од проблемскиот домен
- работат релативно брзо
- даваат приближни резултати
  - не е гарантирано добивање на оптимален резултат

Алчни (greedy) алгоритми



Пронаоѓање на два локални максимуми доколку се тргне лево од точката A или десно од точката A

Алчни (greedy) алгоритми

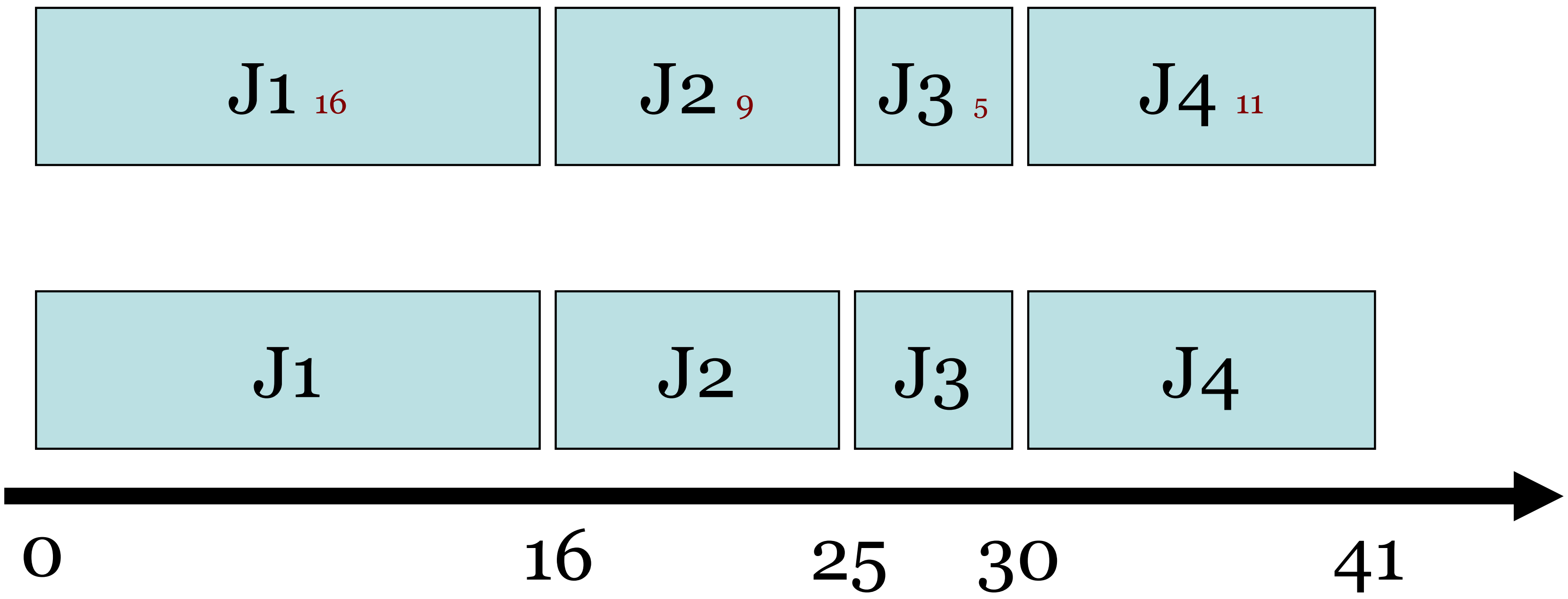
- **Проблем**: Алгоритам за враќање на кусур (пресметка на оптималниот број на банкноти и монети кои треба да се вратат)

Првиот псевдокод кој го предложивме за овој проблем **припаѓа во класата на алчните алгоритми**...

Алчни (greedy) алгоритми

- **Проблем**: operating system schedule
  - Нека се дадени процесите (jobs) $j_1, j_2, \ldots, j_n$, со времиња на извршување $t_1, t_2, \ldots, t_n$, соодветно
  - Процесите треба да се извршат на еден процесор
  - На кој начин тие можат да се распоредат за да се добие најдобро средно време на нивно завршување доколку е добиено истовремено барање за нивно извршување?
  - Да се претпостави дека еднаш кога ќе започне, процесот не смее да се прекине

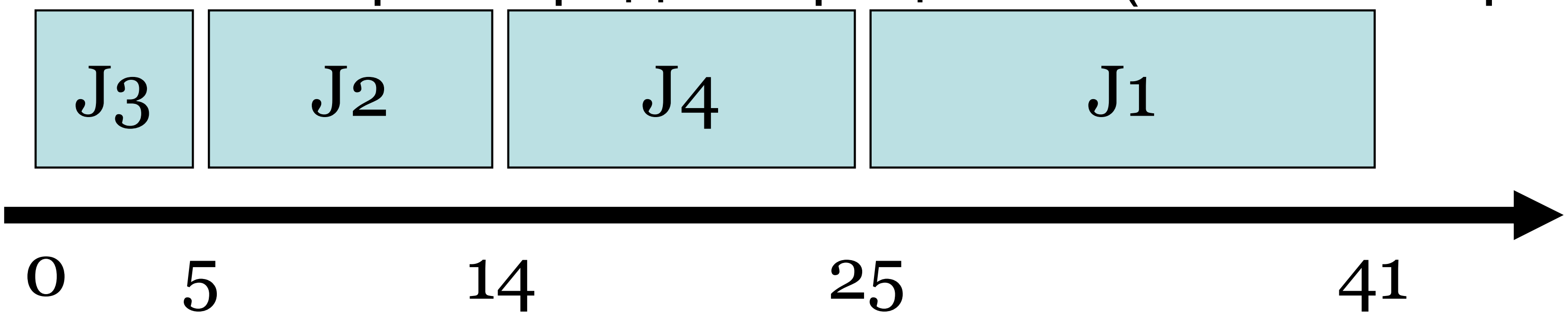
Алчни (greedy) алгоритми



Просечно време
за завршување на процесите
(16+25+30+41)/4=28

Алчни (greedy) алгоритми

Поинаков распоред на процесите (алчен алгоритам)



Просечно време
за завршување на процесите

(5+14+25+41)/4=21,25

Објаснување

- За даден редослед на извршување на процесите $j_{i1}, j_{i2}, \ldots, j_{in}$,

важи дека

- првиот процес j_{i1} завршува за t_{i1}
- вториот процес j_{i2} завршува за $t_{i1} + t_{i2}$
- третиот процес j_{i3} завршува за $t_{i1} + t_{i2} + t_{i3}$
- …

Да обопштиме...

$$C = \sum_{k=1}^{n} (n-k+1) t_{ik}$$

С – цена на чинење

- За претходниот пример, вкупното време за завршување на процесите:

$t_1 + t_1 + t_2 + t_1 + t_2 + t_3 + t_1 + t_2 + t_3 + t_4 = 4\ t_1 + 3\ t_2 + 2\ t_3 + t_4$

Според формулата

- n=4

k=1, (4-1+1) t_1 = 4 t_1

k=2, (4-2+1) t_2 = 3 t_2 $\Rightarrow$ $C = \sum 4t_1 + 3t_2 + 2t_3 + t_4$

k=3, (4-3+1) t_3 = 2 t_3

k=4, (4-4+1) t_4 = t_4

Алчни (greedy) алгоритми

- Основната идеја при градењето на овој распоред е да се овозможи побрзо започнување на процесите, односно нивно пократко чекање

$$C=\sum_{k=1}^{n}(n-k+1)t_{ik} \implies C=(n+1)\sum_{k=1}^{n}t_{ik}\boxed{-\sum_{k=1}^{n}k\cdot t_{ik}}$$

k – реден број на извршување на процес

Заклучок: најдобро е најголемото *k* да се помножи со најголемото времетраење од процесите. Затоа „најкратките“ процеси се извршуваат први, а „најдолгите“ процеси - последни!

Алчни (greedy) алгоритми

- **Проблем**: Huffman кодови (1952)

  - Имаме текстуална датотека составена од карактери кодирани во стандардното множество на ASCII
  - Вкупно постојат 128 симболи
  - За претставување на секој симбол потребни се 7 бита
  - На овие седум бита се додава еден бит за проверка на парноста
  - Доколку имаме C симболи кои треба да се енкодираат на овој стандарден начин потребни се $\lceil \log_2 C \rceil$ бита

Алчни (greedy) алгоритми

симбол	код	честост	вкупно бити
а	000	10	30
е	001	15	45
и	010	12	36
с	011	3	9
т	100	4	12
празно	101	13	39
ентер	110	1	3

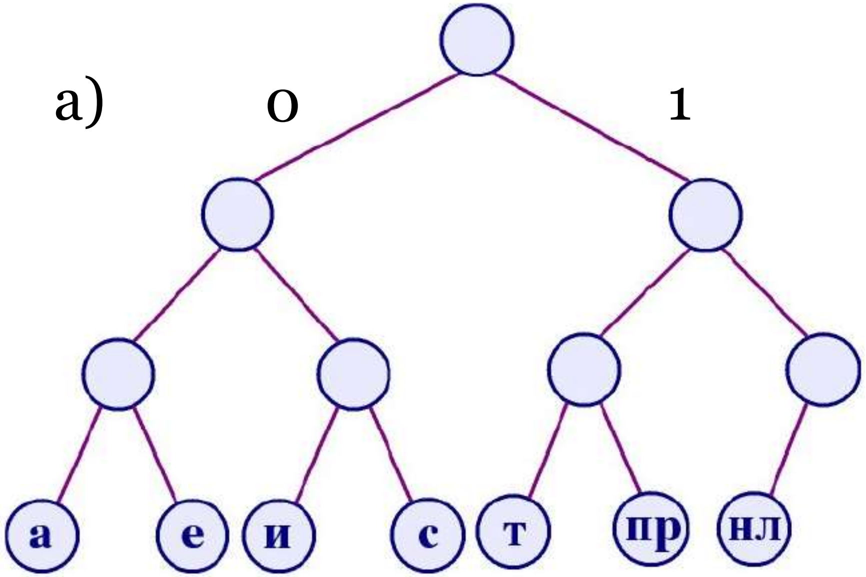
ВКУПНО: 174 бита

Забелешка: во текстуалните датотеки, самогласките се јавуваат почесто од согласките!

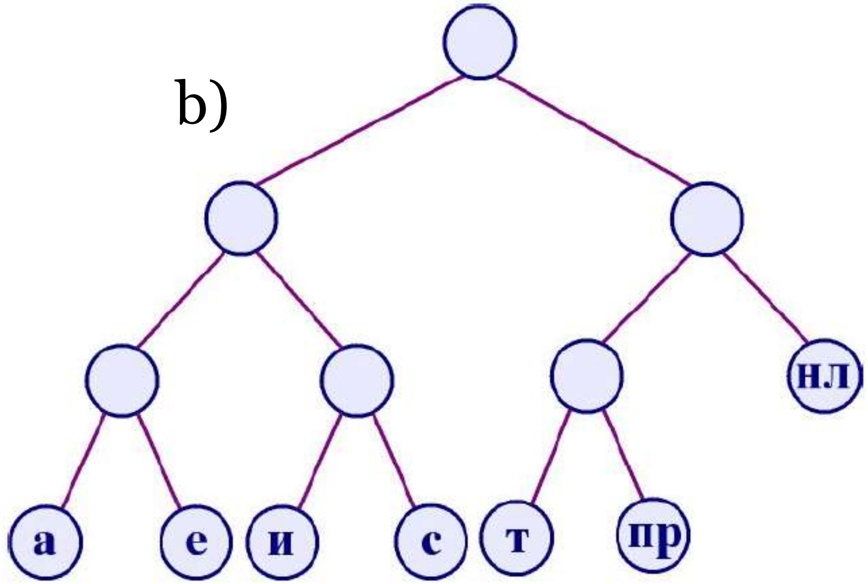
Доколку кодовите на симболите се со променлива должина и притоа кодовите на симболите што се чести се кратки, може да се добие компресија од 25% до 60%

Алчни (greedy) алгоритми

a)



b)

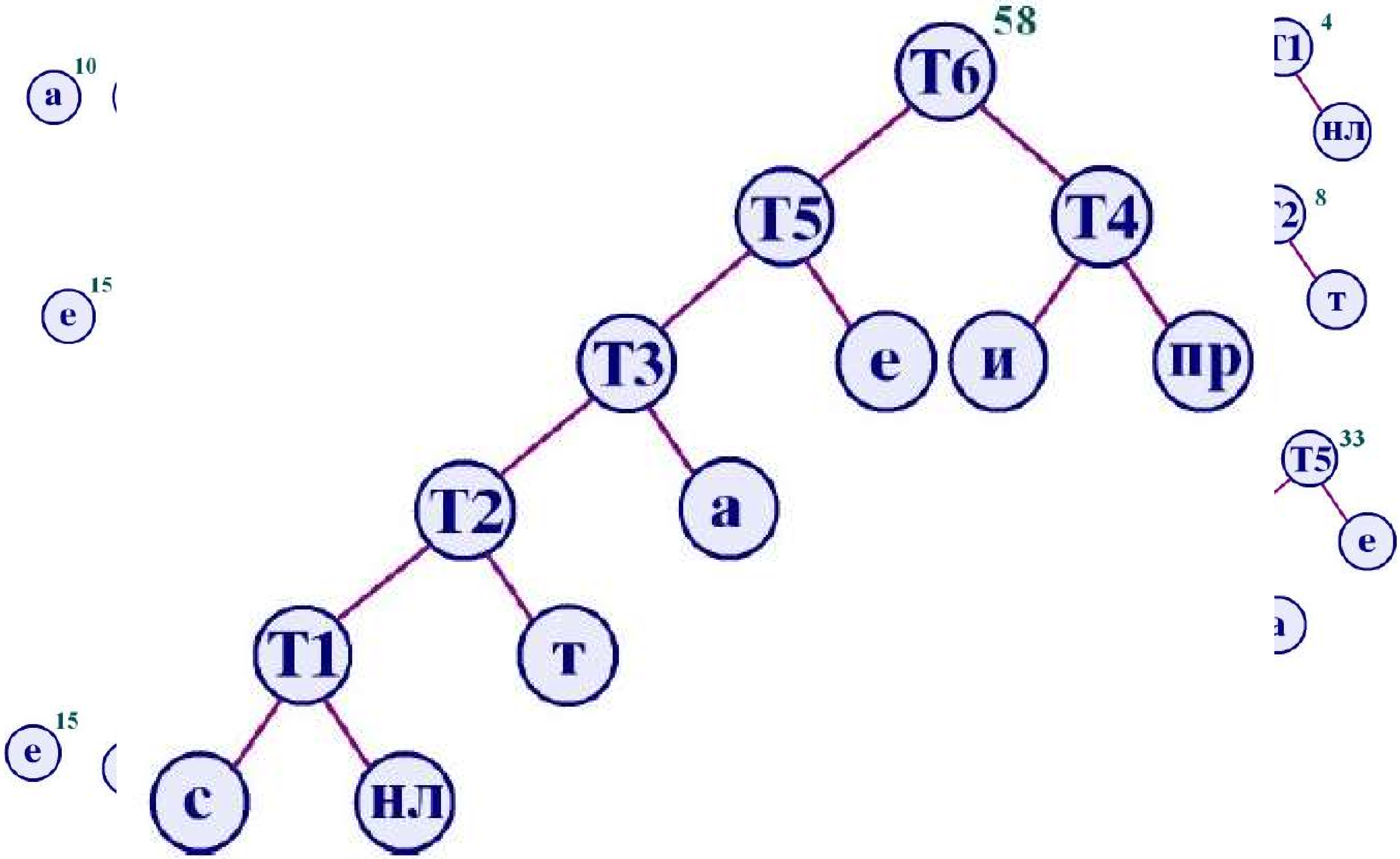


а	000
е	001
и	010
с	011
т	100
празно	101
нов ред	110

а	000
е	001
и	010
с	011
т	100
празно	101
нов ред	11

Што е кодирано во оваа датотека (b): 010011110001011000100111?

Алчни (greedy) алгоритми



Алчни (greedy) алгоритми

- Во секој чекор од алгоритмот се избира најмалиот збир од тежини
- Кодирањето со помош на овој алгоритам се реализира во две фази:
  - утврдување на користените симболи и нивната честота и
  - креирање на стеблото на кодирање
- На почетокот на датотеката треба да бидат дадени кодовите за сите симболи
- Принципот на кодирање може да не е ефикасен за големи датотеки

Алчни (greedy) алгоритми

- **Проблем**: пакување на ранец (Knapsack algorithm)
  - Нека бидат дадени *n* пакети со големини s_1, s_2, ..., s_n, $0 < s_i \leq 1$
  - Проблемот на пакување го бара оптималното пакување на пакетите во ранци со големина 1 (Треба да се искористи најмал број на ранци)
  - Честопати овој проблем се нарекува и проблем на крадецот кој имал торби кој можеле да носат X килограми злато и златни предмети со различни тежини

Алчни (greedy) алгоритми

0,2	0,5	0,4	0,7
0,1	0,3	0,8	

Проблем: Колку ранци со големина 1 се потребни за пакување на овие пакети?

0,2	0,8	
0,3	0,7	
0,1	0,4	0,5

Алчни (greedy) алгоритми

- Постојат две верзии на алгоритмот:
  - **Прва верзија**: не се знае следната големина на пакетот во низата се додека тековниот пакет не се смести во некој ранец
  - **Втора верзија**: познати се сите големини на пакети однапред

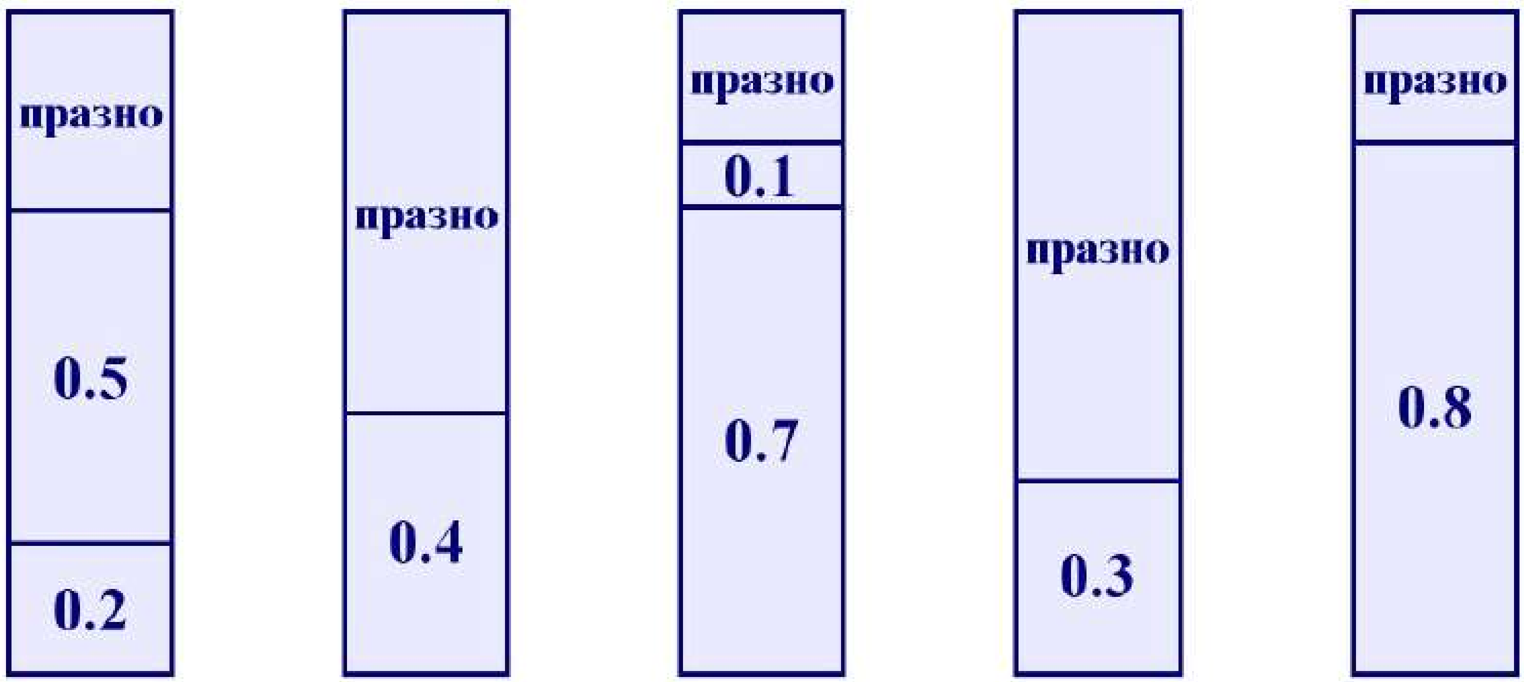
Алчни (greedy) алгоритми

- Next fit решение:
  - Првиот пакет се сместува во првиот ранец
  - При сместување за секој следен пакет се проверува дали има место во ранецот каде што бил сместен претходниот пакет
  - Доколку нема место, се проверува следниот ранец и така натаму се додека не се најде првиот ранец во кој има доволно место каде што пакетот се сместува

Овој алгоритам очигледно работи во линеарно време, но не секогаш дава оптимални резултати

Алчни (greedy) алгоритми

0.2 0.5 0.4 0.7 0.1 0.3 0.8



Во најлош случај овој алгоритам користи двојно повеќе ранци од оптималното решение

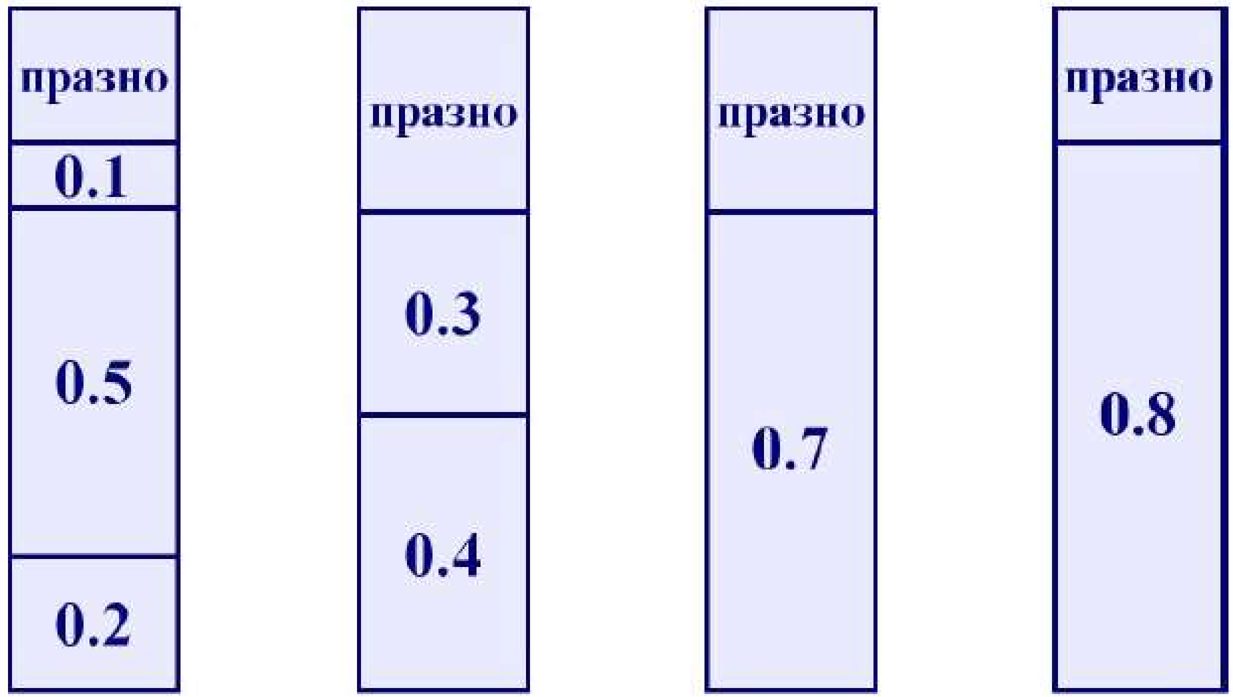
Алчни (greedy) алгоритми

- First fit решение:
  - Првиот пакет се сместува во првиот ранец
  - За секој следен пакет се проверуваат сите ранци од почеток, и пакетот се сместува во првиот ранец кој има простор да го собере
  - Нов ранец се користи само доколку нема простор во предходно започнатите ранци

Овој алгоритам во најдобар случај може да се реализира со комплексност $O(n\log n)$, а во општ случај $O(n^2)$

Алчни (greedy) алгоритми

0.2 0.5 0.4 0.7 0.1 0.3 0.8



Алчни (greedy) алгоритми

- Best fit решение:
  - Првиот пакет се сместува во првиот ранец
  - За секој следен пакет се проверуваат сите ранци од почеток, и пакетот се сместува во ранецот во кој има најмал простор доволен да го собере пакетот со дадена големина
  - Нов ранец се користи само доколку нема простор во предходно започнатите ранци

Алчни (greedy) алгоритми

0.2 0.5 0.4 0.7 0.1 0.3 0.8