

### 3. Многонишково програмиране (Scalable Multithreaded Programming) с „Thread Pools“ (паралелна сортировка )

Съществуват много начини за разпределяне задачите по процесорни ядра или в многопроцесорна среда.

Ще преобразуваме еднонишково приложение към такова, ползващо всички налични изчислителни ресурси.

Ще въведем основни понятия от:

- OpenMP технологията и
- thread pools.

Ползвайки **Visual Studio utilities**, ще демонстрираме измерване на подобренията в производителността.

В началото, ще раздробим общата задача на по-малки, подходящи за нишково разпределяне подзадачи, които бихме могли да подадем към отделните ядра.

Спазваме няколко препоръки при това преобразуване:

- Задачата да подлежи на разпаралеляване.
- Подзадачите да са независими помежду си.
- Да могат да се изпълняват в произволен ред.
- Да поддържат собствени копия на данните си.

## 3.1 Multithreading с OpenMP

OpenMP е една от най-простите технологии за въвеждане на паралелизъм и се поддържа от Visual Studio C++ компилатора след версия 2005. Въвеждането на паралелизма става чрез „OpenMP - pragmas“ директивите в кода. Пример:

```
include <omp.h>                                     // You need this or it won't work
#include <stdio.h>

int main (int argc, char *argv[]) {
    #pragma omp parallel
    printf("Hello World from thread %d on processor %d\n",
           ::GetCurrentThreadID(), ::GetCurrentProcessorNumber() );
    return
}
```

директивата 'pragma' паралелизира следващия я блок код— в случая това е само printf()— и го пуска едновременно по всички изчислителни ресурса. Броят им варира - според инсталираните процесори или ядра.

---

За да разрешите паралелизацията с OpenMP (компиляторът да не игнорира „pragma“ директивите), следва да разрешите OpenMP. За целта: първо

опция към компилатора /openmp  
(Properties | C/C++ | Language | Open MP Support). Второ: **include the omp.h**

Следва пример за разпаралеляване на матрични изчисления (повдигане на степен) и обработка върху всички налични ядра:

```
#pragma omp parallel for  
for (int i = 0; i < 50000; i++)  
    array[i] = i * i;
```

OpenMP притежава и конструктори за контрол на:

- броя създавани нишки;
- Управление последователността на отделните запаралелени блокове;
- Създаване на thread-local data,
  - точки на синхронизация,
  - критични секции и др.

OpenMP е лесен начин за въвеждане на паралелизъм в съществуващ вече код.

OpenMP е проста технология. В много случаи е необходим по-голям контрол над изчислителния процес.

Например, разпределяне нишките по конкретни ядра, динамичен паралелизъм и т.н..

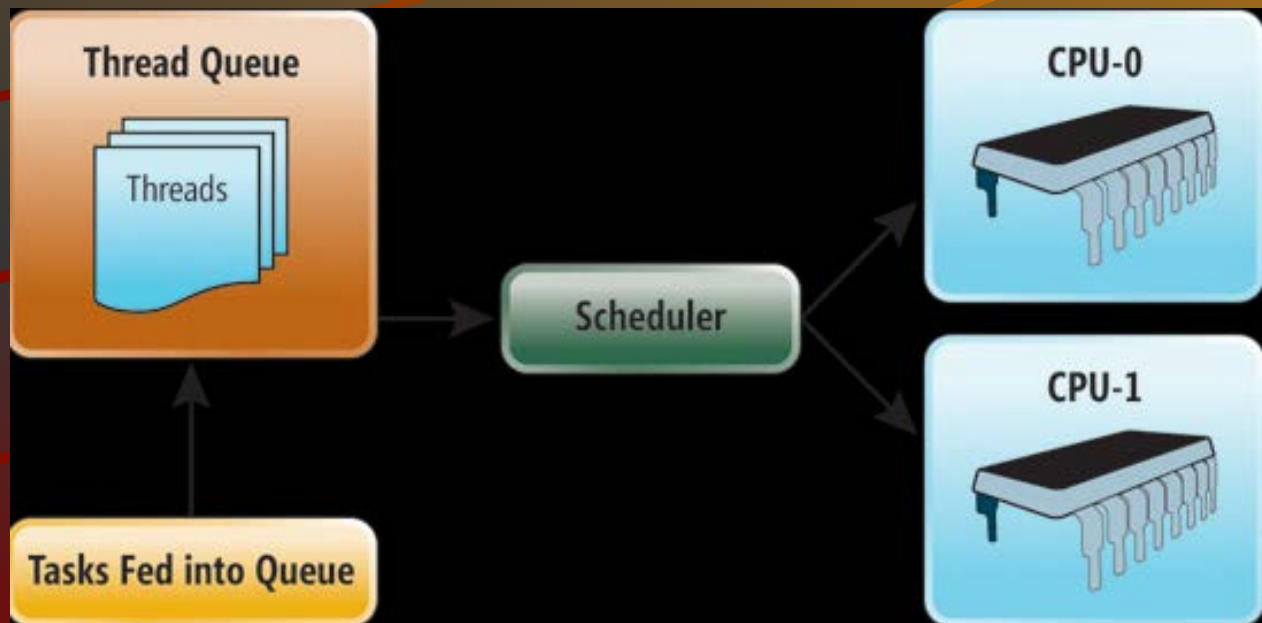
Идват други технологични решения –например, **thread- pools** .

## 3.2 Thread Pool

Нишките се менажират от OS. Те заемат ресурс и създават свое обкръжение. Следователно, честото им създаване и унищожаване е скъпа операция. По-добре е вече създадените да отлежават за повторно използване, при необходимост.

Това място за изчакване се нарича **thread pool** и Windows поема поддръжката му .

Ползването на **thread pool** сваля от програмиста отговорността за често създаване на нишки , унищожаването и управлението им.



Удобно и полезно е да се разпаралели задача в много ядра.



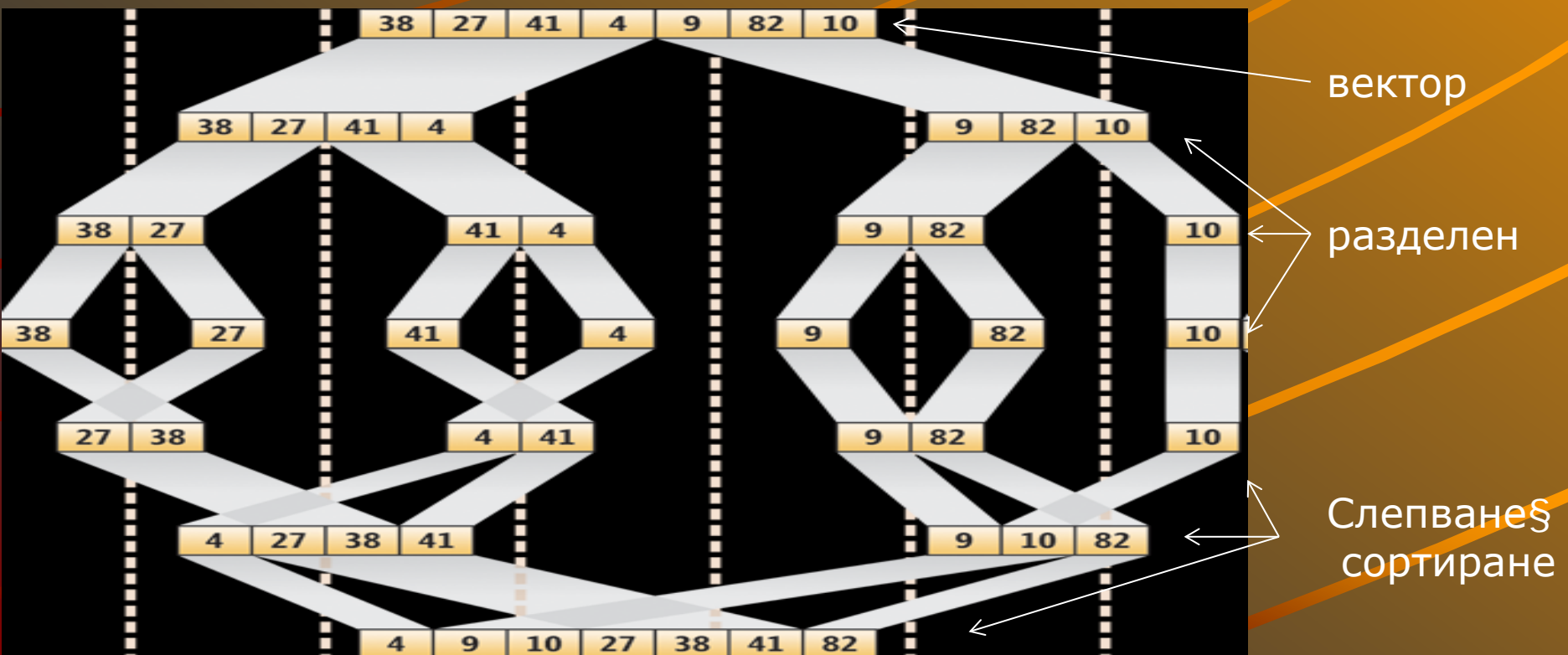
### 3.3 пример с многонишково сортиране

Сортировката не е най-силния алгоритъм за паралелизация. Тук има пречка – как да разпаралеляваме, така че да подсигурир независимост на отделните блокове?

Наивен подход е да заключим достъпа до данните за времето на обработка например чрез mutex, semaphore или критична секция.

По-добро решение е на всяко ядро да се подаде подсекция на масива данни за паралелна сортировка. Този **divide-and-conquer** **подход** е като че ли по-подходящ.

Алгоритмите **merge sort** и **quick sort** работят добре с тази стратегия и тя е подходящо решение за натоварване на многоядрена изчислителна среда.



Разделените подписъци са независими и могат да се подадат към CPU ядрата за паралелна обработка без заключвания.

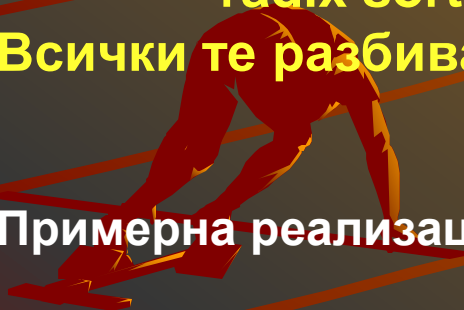
Съществуват много алгоритми за сортировка, податливи на Паралелизация:

- Quick sort,
- selection sort,
- merge sort,
- radix sort

Всички те разбиват данните и ги обработват независимо.

Примерна реализация на сортировка е показана (quick sort на C# ).

Програмата инициализира голям масив с поредица случайни числа и после ги сортира с помощта на quick sort процедура, като отчита и времето за това.



1  
**namespace ParallelSort {**  
**class Program {**

**// For small arrays, use Insertion Sort**

**private static void InsertionSort(**  
**int[] list, int left, int right)**

```
{ for (int i = left; i < right; i++)  
{ int temp = list[i];  
  int j = i;  
  while ((j > 0) && (list[j - 1] > temp))  
  { list[j] = list[j - 1];  
    j = j - 1;  
  }  
  list[j] = temp;  
}}
```

**private static int Partition( int[] array, int i, int j)**

```
{ int pivot = array[i];  
  while (i < j) {  
    while (array[j] >= pivot && i < j)  
    { j--; }  
    if (i < j) { array[i++] = array[j]; }  
    while (array[i] <= pivot && i < j)  
    { i++; }  
    if (i < j) { array[j--] = array[i]; }  
  }  
  
  array[i] = pivot;  
  return i;  
}
```

```
array[i] = pivot;  
return i;  
}
```

2  
**static void QuickSort( int[] array, int left, int right)**  
**{ // Single or 0 elements are already sorted**  
**if (left >= right) return;**

**// For small arrays, use a faster serial routine**  
**if ( right-left <= 32) { InsertionSort(array, left, right);**  
**return; }**

**// Select a pivot, then quicksort each sub-array**  
**int pivot = Partition(array, left, right);**  
**QuickSort(array, left, pivot - 1);**  
**QuickSort(array, pivot + 1, right);**  
**}**

**static void Main(string[] args)**

```
{ const int ArraySize = 50000000;  
  for (int iters = 0; iters < 1; iters++)  
  { int[] array;  
    Stopwatch stopwatch;  
    array = new int[ArraySize];  
    Random random1 = new Random(5);  
    for (int i = 0; i < array.Length; ++i)  
    { array[i] = random1.Next(); }  
    stopwatch = Stopwatch.StartNew();  
    QuickSort(array, 0, array.Length - 1);  
    stopwatch.Stop();
```

```
    Console.WriteLine("Serialt: {0} ms",  
stopwatch.ElapsedMilliseconds);  
..... } } }
```



За да паралелизирате приложението, променете следните оператори:

```
QuickSort( array, lo, pivot - 1);  
QuickSort( array, pivot + 1, hi);
```

Към паралелната реализация:

```
Parallel.Invoke(  
    delegate { QuickSort(array, left, pivot - 1); },  
    delegate { QuickSort(array, pivot + 1, right); }  
);
```

Интерфейсът **Parallel.Invoke** от `Systems.Threading.Tasks` namespace е дефиниран в .NET Task Parallel Library. Той позволява дефиниране на асинхронно изпълнявана функция. В нашата реализация - всяка отделна сортировъчна функция ще се изпълни в отделна нишка.

Макар че е по-добре да запуснете само 1 нишка, а втория подписък да се поеме за сортиране от текущата, реализация с 2 запуснати нишки за сортировка е по-симетрична и илюстрира колко лесно е преобразуването на серийна програма в паралелен еквивалент.

Разумен въпрос е: паралелизацията подобри ли производителността?

Visual Studio 2010 включва няколко tools за такава оценка :

**Visual Studio 2010 Performance Wizard** мери производителност в паралелна реализация по време на изпълнение.

Можете да наблюдавате и core utilization.

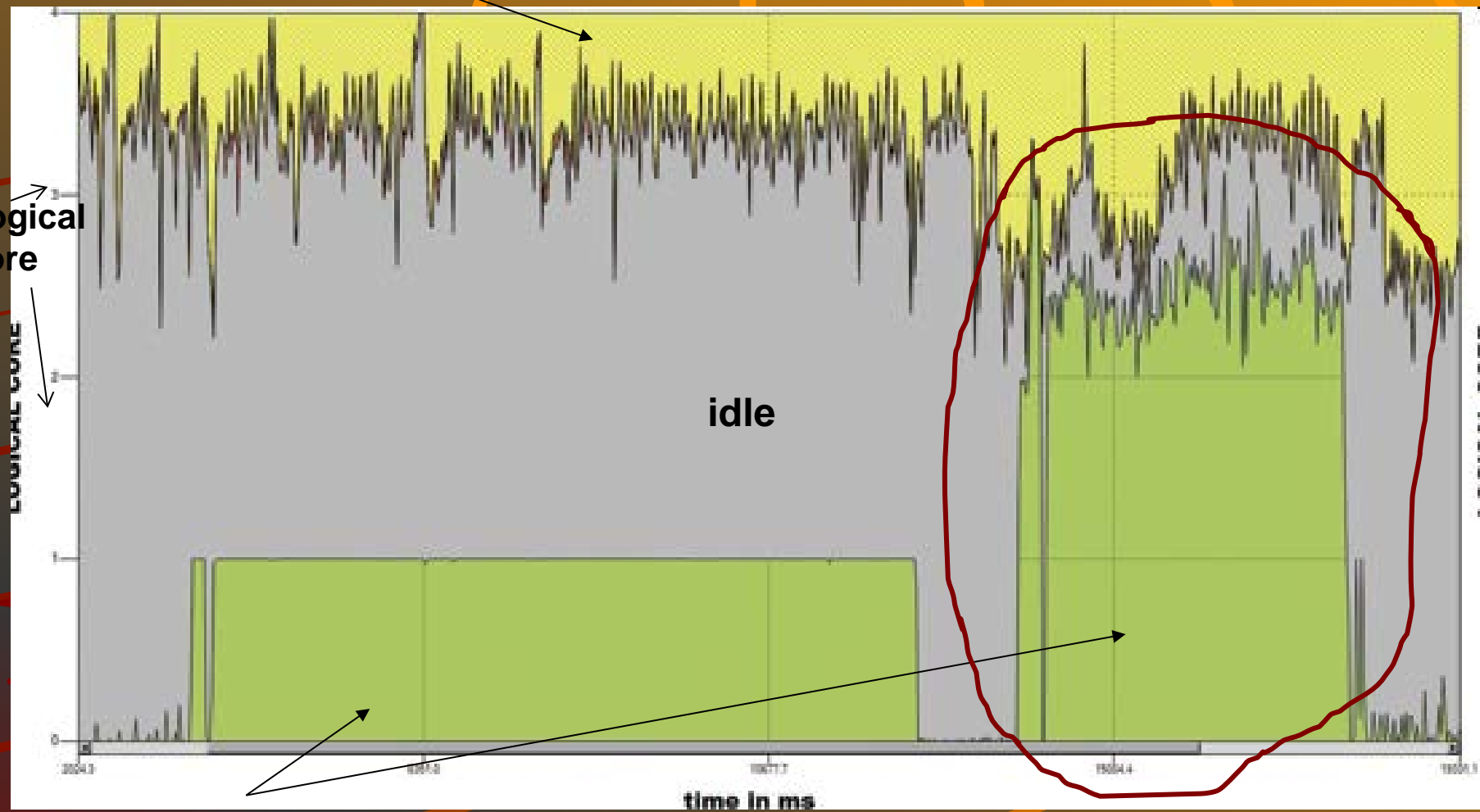
Нека тестовата програма стартира серийна сортировка, заспива секунда и тогава стартира паралелна версия на сортировъчния алгоритъм.  
За 4-ядрен компютър , получваме следната графична представа:

В началото е видно натоварването при 1 ядрена реализация – 100% използваемост на единственото ядро. Впоследствие се вижда **2.25кратно ускорение**.

Паралелната реализация ускорява с около 45% в сравнение със серийния си еквивалент.

OS и други програми

Logical  
core



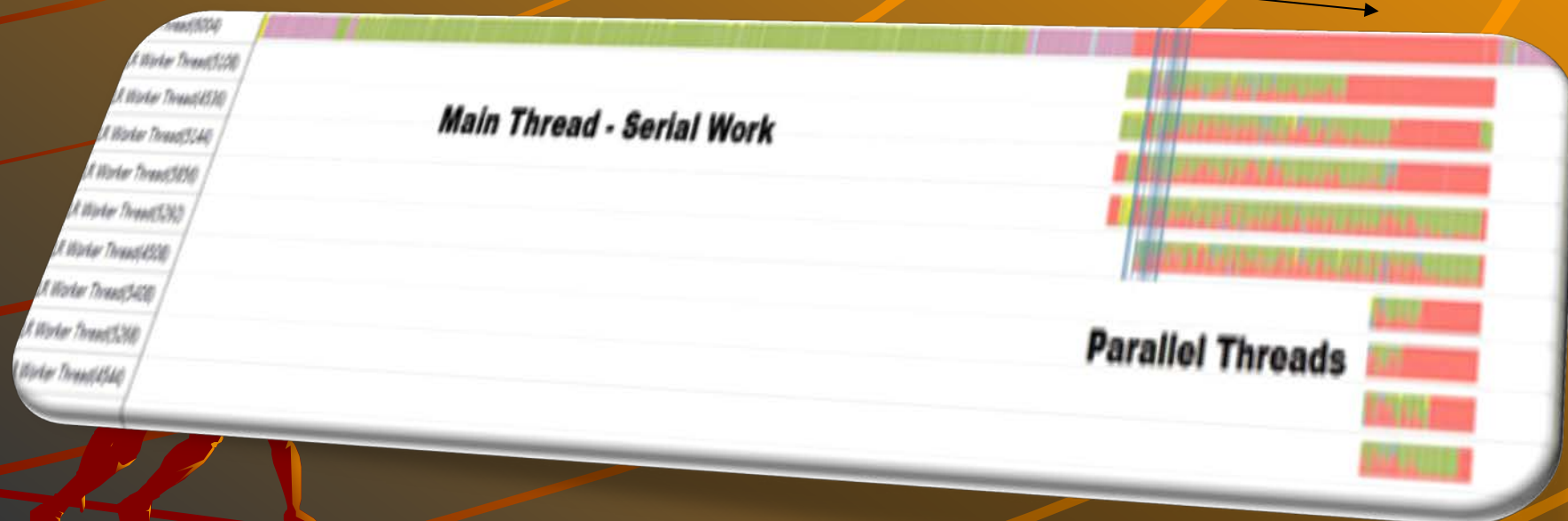
приложението

## Друга визуализация

Показваме как приложението ползва наличните нишки.

Една нишка работи почти цялото време.

Розовият цвят показва нишки, които са блокирани от други.



Вижда се , че макар натоварването на CPU ядра е подобро, има потенциал за още значителни оптимизации.


## 4. Динамично програмиране

Рекурсията не е оптималната техника → в ред случаи рекурсивният алгоритъм следва а се пренапише в нерекурсивен вариант със запомняне междинни резултати в таблица.

Това е “динамичното програмиране”.

### 4.1 таблици вместо рекурсия

Ето неефективен вариант за изчисляване числата на Фибуначи:



```
int fib( in n)
{
    if( n <= 1 )
        return 1;
    else
        return fib( n - 1 ) + fib( n - 2 );
}
```


$$T(N) \geq T(N-1) + T(N-2)$$

Вече изследвана в курса сложност: експоненциална



Можем, обаче, да съхраним изчисленията вече  $F_{n-1}$ ;  $F_{n-2}$ . :

```
int fibonacci( int n )
{
    if( n <= 1 )
        return 1;
    int last = 1;
    int nextTolast = 1;
    int answer = 1;
    for( int i = 2; i <= n; i++ )
    {
        answer = last + nextTolast;
        nextTolast = last;
        last = answer;
    }
    return answer;
}
```



Имаме  $O(N)$ .

Ако горната модификация не е направена, за изчисление на  $F_n$  вика рекурсивно  $F_{n-1}$  и  $F_{n-2}$ .....



## 4.2 оптимално бинарно търсене в дърво

Имаме списък думи  $w_1, \dots, w_n$  с вероятности на появяване:  $p_1, \dots, p_n$

Искаме да подредим думите в бинарно дърво, така че общото време за намиране на дума да е минимално.

В бинарно дърво, за да стигнем до елемент с дълбочина  $d$ , правим  $d+1$  сравнения. Така че, ако  $w_i$  е на дълбочина  $d_i$ , ние искаме да минимизираме

$$\sum_{i=1}^N p_i (1 + d_i).$$

дума	вероятност
------	------------

a	0.22
---	------

am	0.18
----	------

and	0.20
-----	------

egg	0.05
-----	------

if	0.25
----	------

the	0.02
-----	------

two	0.08
-----	------



Друга подредба: балансирано, бинарно д-во



Някаква Greedy стратегия:  
^ вероятност → ляв клон



Оптимално,  
но неинтуитивно д-во

първото използва greedy стратегия. Най-вероятната дума е най-горе.  
 Второто е балансирано search tree (в дясно с по-ниско  $p_i$ ).  
 Третото е оптималното – виж таблицата:

вход дума	вероятност	дърво 1		дърво2		дърво3	
		дълбочина	цена	дълбочина	цена	дълбочина	цена
a	0.22	2	0.44	3	0.66	2	0.44
am	0.18	4	0.72	2	0.36	3	0.54
and	0.20	3	0.60	3	0.60	1	0.20
egg	0.05	4	0.20	1	0.05	3	0.15
if	0.25	1	0.25	3	0.75	2	0.50
the	0.02	3	0.06	2	0.04	4	0.08
two	0.08	2	0.16	3	0.24	3	0.24
общо	1.00		2.43		2.70		2.15

сравнение на 3 дървета с бинарно търсене

При оптимално, бинарно „search tree“ (формирано по алгоритъм, подобен на този на Huffman) данните не са само в листата (както беше при Huffman компресията) и следва да удовлетворяваме критерий за binary search tree.

Поставяме сортирани според някакъв критерий думи  $W_{\text{left}}, W_{\text{left}+1}, \dots, W_{\text{right}-1}, W_{\text{right}}$  в бинарно дърво. Нека сме постигнали оптималното бинарно дърво в което имаме корен  $W_i$  и под-дървета за които :  $\text{Left} \leq i \leq \text{Right}$

Тогава лявото поддърво съдържа елементите  $W_{\text{left}}, \dots, W_{i-1}$ , а дясното –  $W_{i+1}, \dots, W_{\text{right}}$  ( критерий за binary search tree ).

Поддърветата също правим оптимални. Тогава можем да напишем формулата за цената  $C_{\text{left}, \text{right}}$  на оптимално binary search tree. Лявото поддърво има цена  $C_{\text{left}, i-1}$ , дясното –  $C_{i+1, \text{right}}$ , спрямо корена си.





## Структура на оптимално за претърсване, бинарно дърво

търсим минимизация за цената на цялото дърво (т.е. кое „i“ е в корена):

$$C_{Left, Right} = \min_{Left \leq i \leq Right} \left\{ p_i + C_{Left, i-1} + C_{i+1, Right} + \right.$$

$$\left. \sum_{j=Left}^{i-1} p_j + \sum_{j=i+1}^{Right} p_j \right\}$$

Спрямо това i

$$= \min_{Left \leq i \leq Right} \left\{ C_{Left, i-1} + C_{i+1, Right} + \sum_{j=Left}^{Right} p_j \right\}$$

За да отчетем че всеки възел в тези поддървета е 1 ниво по-дълбоко спрямо общия корен

На основата на тази формула се гради алгоритъмът за минимална цена (оптималното дърво).

Търсим  $i$ , така че да се минимизира  $C_{Left, Right}$ .

Таблицата е резултатът от работата на алгоритъма:

a..a	am..am	and..and	egg..egg	if..if	the..the	two..two
.22 a	.18 am	.20 and	.05 egg	.025 if	.02 the	.08 two

a.am	am..and	and..egg	egg..if	if..the	the..two
.58 a	.56 and	.30 and	.35 if	.29 if	.12 two

a.. and	am..egg	and..if	egg..the	if..two
1.02 am	.66 and	.80 if	.39 if	.47 if

a..egg	am.. if	and..the	egg..two
1.17am	1.21 and	.84 if	.57 if

a.. if	am.. the	and..two
1.83 and	1.27 and	1.02 if

a.. the	am..two
1.89 and	1.53 and

a.. two
2.15 and

цена

Корен на оптималното (до момента ) бинарно д-во

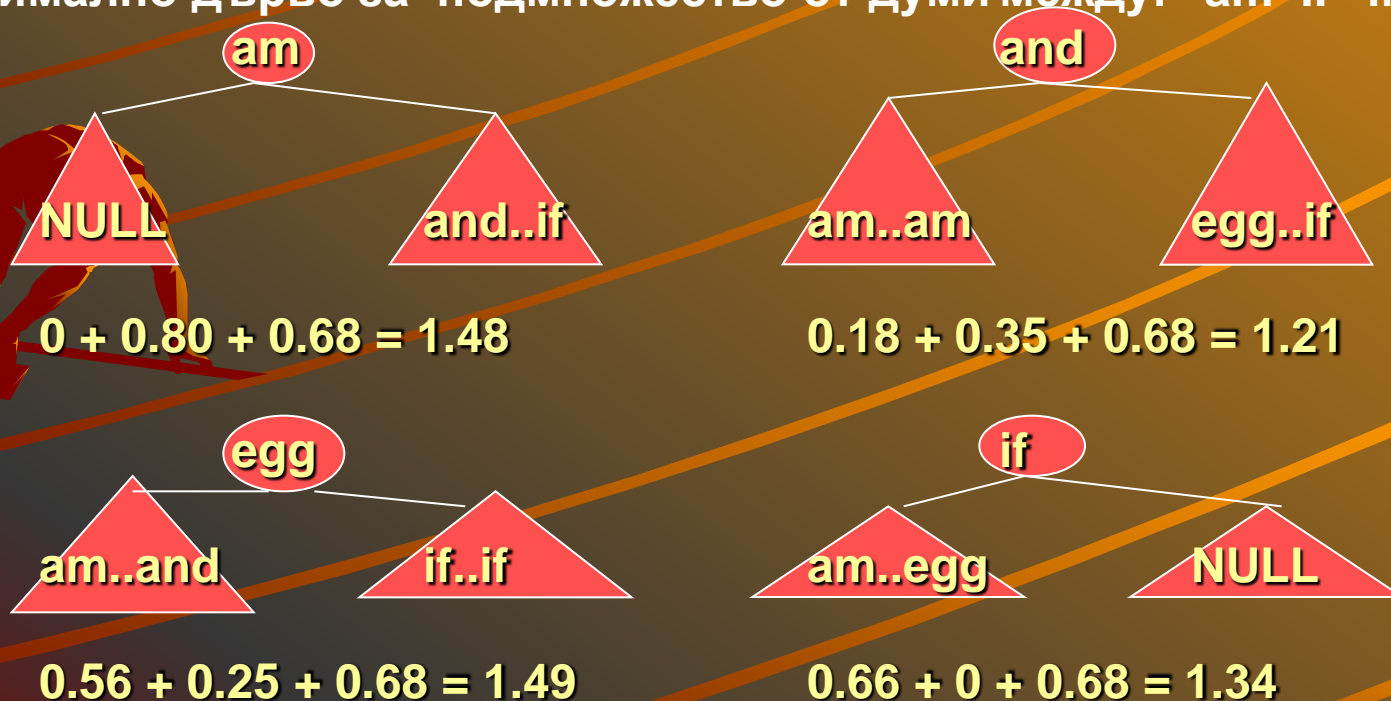
Определянето на цената на това поддърво ( за подредица от am до if) е показано на следващ слайд

Цена / корен на оптимално д-во, съдържащо всички думи (третият граф)

за всяка подредица от думи се пази и цена ( в случая вероятност) и корен на оптималното (до момента) binary search tree.

Най-долу се получава цена и корен за оптималното дърво, съдържащо всички думи (това е и оптималното дърво, показано трето в графите).

Как са изчислени стойностите за всички възможни корени до откриване на оптимално дърво за подмножество от думи между: am .. if :



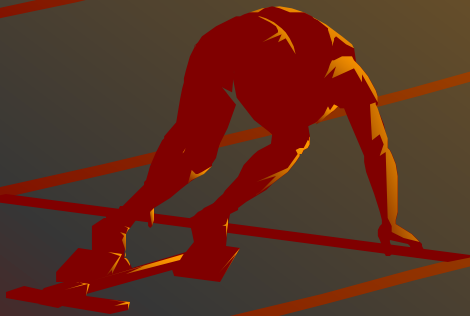
Последователно се поставят за корени думите: am, and, egg, if  
и се изчислява цена за дървото - оттам откриваме минималната цена.

Например, когато and е корен, лявото поддърво am...am има цена 0,18  
(вече определена), дясното – egg-if с цена 0.35 и

$$p_{am} + p_{and} + p_{egg} + p_{if} = 0.68 \quad (\text{винаги})$$

Тогава общата цена е 1,21

Времето е  $O(N^3)$  защото имаме тройно вложен цикъл



### 4.3\* Най-къс път м/ду всички двойки точки в граф

Имаме насочен граф  $G=(V,E)$ .

Алгоритъмът на Dijkstra намиращ най-къси пътища от произволен връх до всички останали ( $O(|V|^2)$ ) работеше на етапи: започваме от връх  $s$ ; всеки връх ( $v$ ) се селектира като междинен. За всеки връх  $w$ -дестинация, дистанцията  $d$  се изчислява така:

$$d_w = \min (d_w, d_v + c_{vw})$$

Сега селектираме върхове последователно. Нека  $D_{k,i,j}$  е теглото на най-късия път от  $v_i$  до  $v_j$  през  $v_1, v_2 \dots v_k$  ( $k$  междинни върха) .

Т.е.:  $D_{0,i,j} = C_{i,j}$  или безкрайност ако няма ребро.

$D_{|V|,i,j}$  е най-къс път от  $v_i$  до  $v_j$  из целия граф.

Когато  $k > 0$ , можем да напишем оптимизирана програма за най-къс път

$D_{k,i,j}$ .

Най-късият път е или този който изобщо не минава през  $v_1, v_2 \dots v_k$  междинно, или получен от сливане на 2 пътя:  $v_i \rightarrow v_k$ ;  $v_k \rightarrow v_j$ , всеки минаващ през първите  $k-1$  междинни върхове .



```
for (int k=0;k<n; k++)  
    // нека всеки връх разгледаме като междинен  
    for (int l=0;l < n; l++)  
        for(int j=0; j < n; j++)  
            if(d[ i ][ k ] +d[ k ][ j ] < d[ i ][ j ];  
            {  
                // нов най-къс път  
                d[ i ][ j ] = d[ i ][ k ] + d[ k ][ j ];  
                path[ i ][ j ] = k;  
            }
```

или формулата:

$$D_{k,l,j} = \min\{D_{k-1,i,j}, D_{k-1,i,k} + D_{k-1,k,j}\}$$

3

Времето отново е  $O(V^3)$

Освен това алгоритъмът е отличен за разпаралеляване.

## 5. Алгоритми с backtracking

Достига до добри решения , но е непредвидимо бавен, поради връщанията

Пример: подреждане на мебели в къща.  
спира се на различни етапи и се кара до изчерпване възможностите.

4.1 пример: проблем – реконструкция (приложение във Физика, молекулярна биология..)

Нека имам  $N$  точки:  $p_1, p_2, \dots, p_N$  подредени по оста  $x$ .

Дадени разстоянията : на брой  $N(N - 1) / 2$ . Да приемем:  $x_1 = 0$

Ако имахме дадени координатите, лесно можем да изчислим

разстоянията. Оценката е  $O(N^2)$

Нека дистанциите с дадени сортирано.

Задачата е: да се реконструират координатите на точките от дистанциите между тях.

дадени:  $D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10\}$

D има 15 елемента, следователно,  $N=6$

нека координатата  
очевидно

$$x_1 = 0$$

$$x_6 = 10$$



$$D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8\}$$

най-голямата оставаща дистанция е 8, следователно, или  $x_2=2$  или  $x_5=8$ .  
Откъдето и да тръгнем, все ще стигнем края – ако съществува (връщанията от неуспех са равновероятни).

Нека  $x_5=8$ .

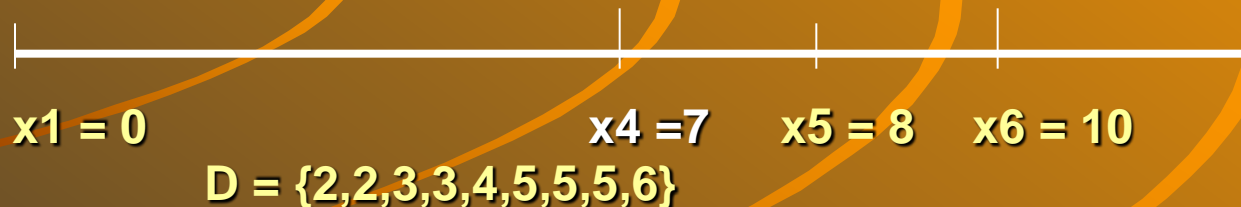
$$\text{тогава } x_6 - x_5 = 2; x_5 - x_1 = 8$$



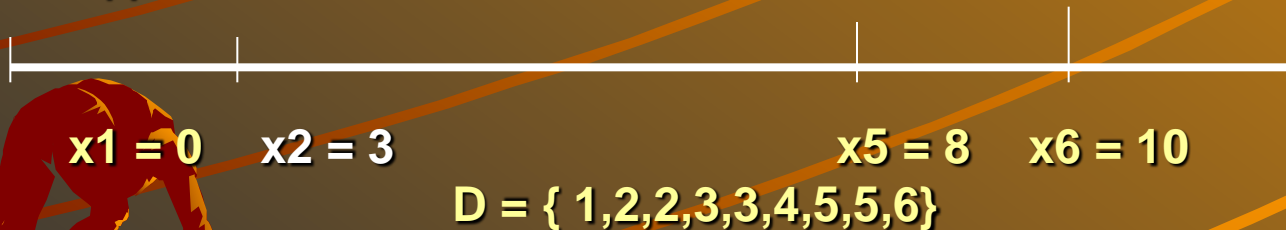
$$D = \{1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7\}$$

Сега 7 е най-голямото оставащо разстояние. Или  $x_4=7$  или  $x_2=3$  (за да има разст. = 7 от  $x_2$  до  $x_6$ ).  
Ако  $x_4 = 7$  – проверяваме разст. до  $x_6$  и  $x_5$  и виждаме че ги има във вектора с разст.  
Ако решим  $x_2=3$ , то  $x_3-x_1=3$ ;  $x_5-3=5$  - също са във вектора. Значи и това става.

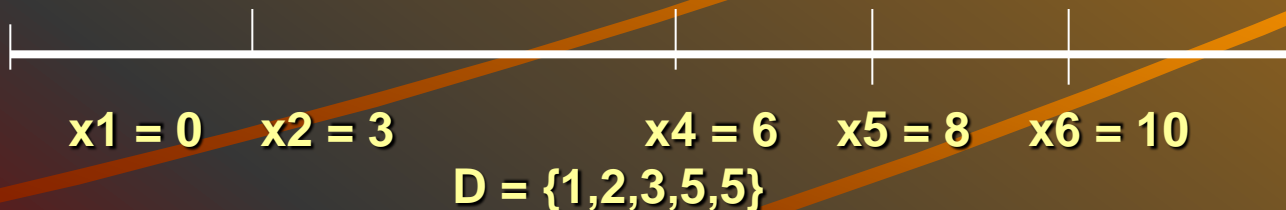
Избираме едното за да продължим. Нека  $x_4=7$



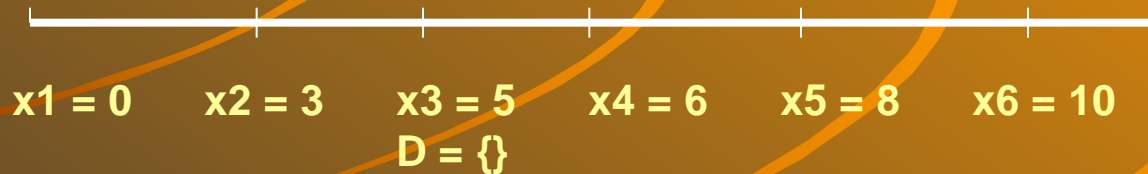
Сега макс. разст. е 6, така че или  $x_3=6$  или  $x_2=4$ . Проверяваме  $x_3=6$  - не може. Проверяваме  $x_2=4$  - не може. **Връщаме се назад.**  
 $x_4=7$  отпадна. Сега опитваме  $x_2=3$



сега остава да изберем или  $x_4=6$  или  $x_3=4$ .  $x_3=4$  е невъзможно.  
Значи остава  $x_4=6$ :

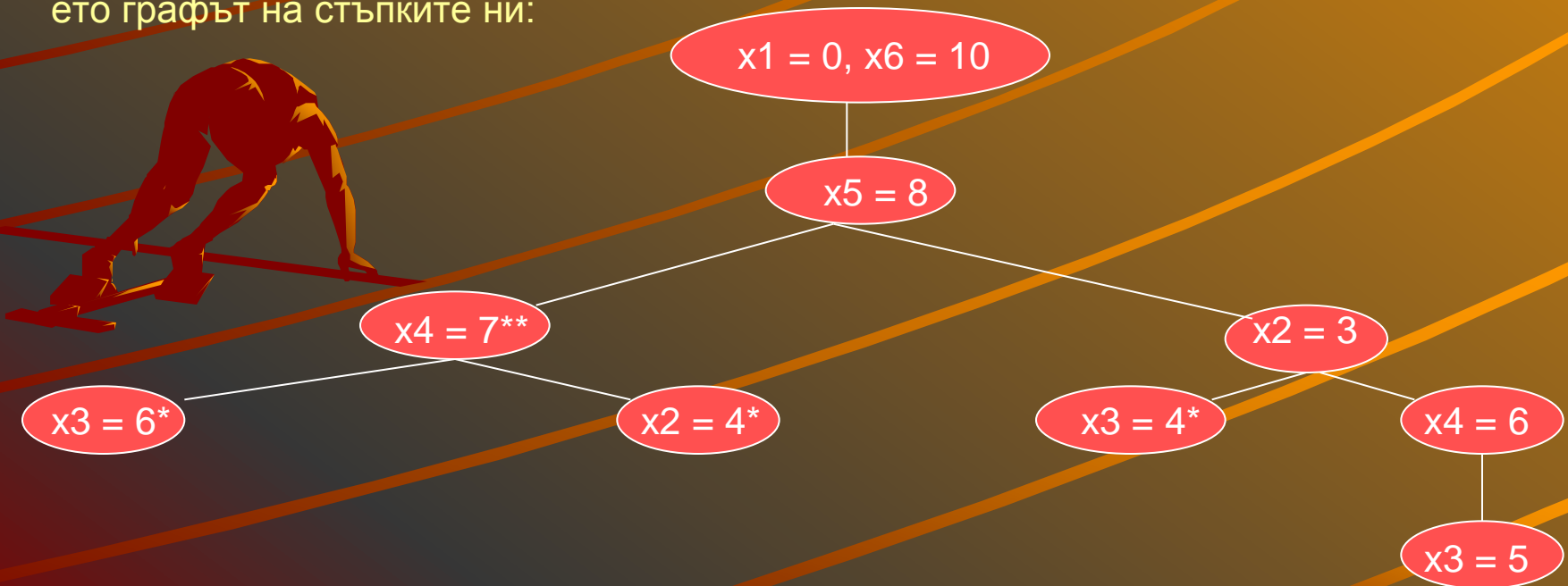


остана  $x_3=5$



достигнахме успешен край! Векторът от разстояния се изпразни

ето графът на стъпките ни:





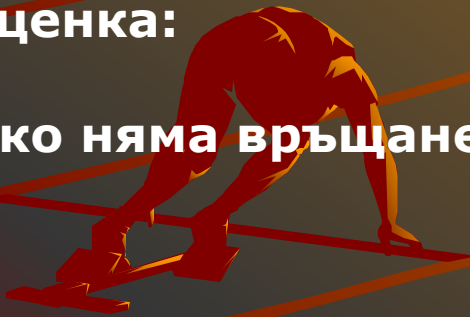
**Забележки:**

- \* избраното решение води до разстояния, които липсват в D.**
- \* \* върхът има само неуспяващи деца, следователно този път се изоставя.**

**Поради backtracking анализът е вероятностен от... до...**

**оценка:**

**Ако няма връщане  $O(N^2 \log N)$**



## 5.2 Елементи от теория на игрите

1.изчерпателен анализ на ходовете – възможно ли е?

Примери : компютърен шах, игра: крави /бикове  
крави/бикове е лесна за програмиране, защото броят е изчерпаем, известни са слабите ходове (могат да се вкарат в таблица). Алгоритъмът никога не греши (никога не губи) и винаги ще спечели ако му се даде възможност.

### Стратегия МИНИ/макс

разглеждаме “силата” на позициите: Ако води победа – оценката е +1, ако равенство: 0; ако е позиция с която компютърът губи: -1.

Такива позиции са терминални.

За останалите → рекурсивно се играе по възможните ходове.

Значи стратегията е : противникът се стреми да минимизира стойността (някаква оценъчна функция) ,

Играчът (компютърът) – да я максимизира.

Пробват се всички възможни в момента ходове.Избира се този с макс. стойност. За ход на противника – същото: прохождат се всички ходове и се избира този с минимизираща стойност.

```

int TicTakToe:: findCompMove( int & bestMove)
{ int i, responseValue;
  int dc // don't care: стойността не се използва
  int value;
  if( fullBoard () )
      value = DRAW; //равенство
  else
  if( immediateCompWin( bestMove ) ) //има възможност за редица
      return COMP_WIN
  else
      { value = COMP_LOSS; bestMove = 1;
        for( i = 1; i < 9; i++ ) // пробва се с всяко квадратче
        {
            if( isEmpty( i ) )
            {
                place( i, COMP );
                responseValue = findHumanMove( dc );
                unplace( i );

                if( responseValue > value ) // търси подобрене
                {
                    //update best move
                    value = responseValue;
                    bestMove = i;
                }
            }
        }
      }
  return value;
}

```



ето играта на човека (двете функции рекурсивно се викат, като оценките им за успешен ход са противоположни):

```
int TicTacToe::findHumanMove( int & bestMove)
```

```
{    int i, responseValue;
```

```
    int value; int dc //don't care
```

```
    if( fullBoard())
```

```
        value = DRAW;
```

```
    else
```

```
        if( immediateHumanWin( bestMove) )
```

```
            return COMP_LOSS;
```

```
    else
```

```
        { value = COMP_WIN; bestMove = 1;
```

```
        for( i = 1; i <= 9; i++ ) //изпробва всяко квадратче
```

```
        {    if( isEmpty( i ) )
```

```
            {
```

```
                place( i, HUMAN);
```

```
                responseValue = findCompMove( dc );
```

```
                unplace( i ); //restore board
```

```
                if( responseValue < value )
```

// търси намаляване

```
                {    // update best move
```

```
                    value = responseValue;
```

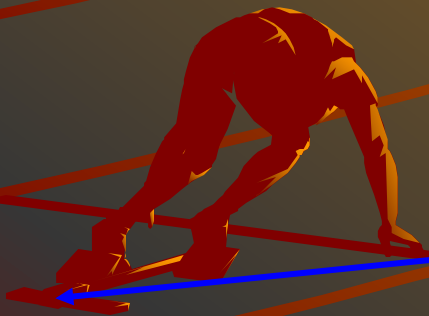
```
                    bestMove = i;
```

```
                }}
```

```
        }} return value;
```

```
}
```

2/2



Най- много изчисления са нужни когато компютърът започва играта (много възможности за проверка). Избира се позиция 1 (горе , ляво, макар че това е условно). Има 97,162 възможни продължения за анализ.

Ако компютърът играе втори – позициите за анализ са намалели на 5185. Ако е бил избран центърът: 9761, при избор на ъгъл: 13,233.

-----

Очевидно при шах тази стратегия за прохождение ( до терминален ) е непосилна ( повече от  $10^{100}$  позиции за анализ ). Спира се донякъде, вика се функция, оценяваща достигнатата позиция. (напр. проходимост на фигури, качество на фигури и т.н.) .  
Тази функция е основната за шах- програмата.

Много е важно колко хода напред могат да се прегледат (дълбочината на рекурсията). За да се увеличат:

в таблица се пазят вече анализирани (и аналогични) ходове. Когато се срещнат – се скача към терминален.



$\alpha - \beta$  окастрияне (pruning) (подобрение в мини/макс стратегиите)

Ето „game tree“ за някаква игра:

max

min

max

min

Ето същото дърво, но орязано. Стойността в корена е същата.. Как?

Max

min

max

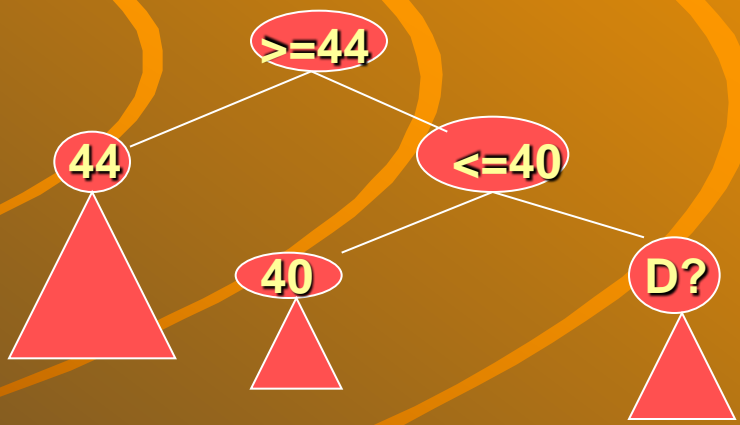
min

Каквото и да е D, не е важно. Ето събраната до момента информация, която ще служи за изчисляване на D:

Ето интересуваща ни извадка:

Вземаме max

вземаме min

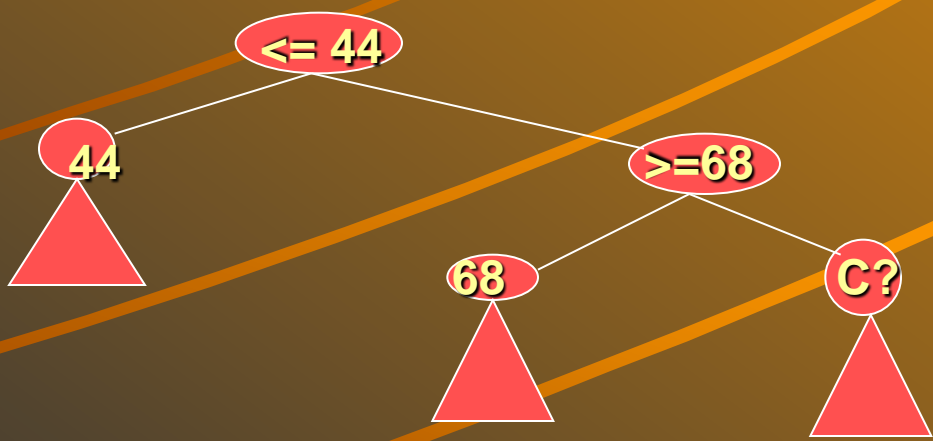


$\alpha$  Окастрияне.

Обратният случай:

min

max



$\beta$  Окастрияне.

Стратегията е лесна за реализация и силна. Колкото по-нагоре в дървото я приложим , толкова ефектът е по-добър.

Ограничава търсенето до  $O(\sqrt{N})$  възли, където  $N$  е размера на цялото дърво. Това позволява удвояване броя на прохожданите ходове.

Играта крави/бикове не е подходящ пример за ползата ( има много идентични (като оценка ) ходове). Въпреки това прилагането ѝ в началния момент свежда от 97162 до 4493 възлите за разглеждане (само нетерминални възли се разглеждат при тази стратегия).

Ето алгоритъма с добавена стратегия  $\propto \beta$  окастриране:

```
Int TicTacToe::findCompMove( int &bestMove, int alpha, int beta )
```

```
{ int i, responseValue;
```

```
int dc
```

```
int value;
```

```
if( fullBoard() )    value = DRAW;
```

```
else
```

```
    if( immediateCompWin( bestMove)
```

```
    return COMP_WIN; //bestMove се устан. om immediateCompWin()
```

```
    else
```

```
        {    value = alpha; bestMove = 1;
```

```
        for( i = 1; i <= 9 && value < beta; i++)
```

```
        { if( isEmpty( i ) )
```

```
            {
```

```
                place ( i, COMP );
```

```
                responseValue = findHumanMove( dc );
```

```
                unplace( i ); //restore board
```

```
                if( responseValue > value )
```

```
                { // update best move
```

```
                    value = responseValue;
```

```
                    bestMove = i;
```

```
                }
```

```
            }
```

```
        }
```

```
    }
```

```
    return value;
```

```
}
```

Преди повикване:

alpha = COMP\_LOSS

beta = COMP\_WIN

