ПОДАТОЧНИ СТРУКТУРИ И АНАЛИЗА НА АЛГОРИТМИ

ЧАС 5: ТЕХНИКИ ЗА КРЕИРАЊЕ НА АЛГОРИТМИ

АУДИТОРИСКИ ВЕЖБИ

БОЈАНА ВЕЛИЧКОВСКА

ТЕХНИКА НА ГРУБА СИЛА (BRUTE FORCE)

- Алгоритмот дизајниран според техника на груба сила наоѓа решение на даден проблем барајќи елемент(и) со определено **својство**. Најчесто пребарувањата се извршуваат над **комбинаторни** објекти (пермутации, комбинации, подмножества на множество, итн.)
  - **Метод**:
    - Креирајте листа од **сите** **потенцијални решенија** на даден проблем
    - Оценувајте ги решенијата, **едно по едно**, отфрлајте ги неприфатливите, а чувајте го до тогаш најдоброто решение
    - Кога пребарувањето ќе заврши, **вратете** го **најдоброто** решение

ТЕХНИКА НА ГРУБА СИЛА (BRUTE FORCE)

- (+)
  - Широка применливост
  - Едноставност
  - За дадени проблеми дава разумни решенија (множење на матрици, сортирање, пребарување,...)
  - Секогаш го враќа точното и најоптимално решение
- (-)
  - Ретко дава ефикасни алгоритми (троши мемориски и временски ресурси)
  - Некогаш е неприфатливо бавен

ТЕХНИКА НА ГРУБА СИЛА (BRUTE FORCE)

- **Задача**: Најдете го минималното растојание помеѓу две точки (во множество од n точки).
- **Решение**: Пресметајте ги растојанијата помеѓу секој пар точки и вратете го минималното растојание и координатите на точките кои го формираат истото.

```
class Tochka {
public float x;
public float y;
}
```

ТЕХНИКА НА ГРУБА СИЛА (BRUTE FORCE)

```
public static float min_rast(Tochka [ ] p, int n) {
int i,j;
float pom;
int [][] minkoord = new int [2][2];
float min= Math.sqrt(Math.pow((p[0].x-p[1].x),2)+Math.pow((p[0].y-p[1].y),2));
minkoord[0][0]=p[0].x; minkoord[0][1]=p[0].y; minkoord[1][0]=p[1].x; minkoord[1][1]=p[1].y;
for (i=0;i<(n-1);i++)
for (j=i+1;j<n;j++)
{pom=Math.sqrt(Math.pow((p[i].x-p[j].x),2)+Math.pow((p[i].y-p[j].y),2));
if(pom<min)
{min=pom; minkoord[0][0]=p[i].x; minkoord[0][1]=p[i].y; minkoord[1][0]=p[j].x; minkoord[1][1]=p[j].y;
}}
return min;
}
```

Комплексност => $O(n^2)$
Како да се забрза алгоритмот? =>
употребете раздели и владеј стратегија

ТЕХНИКА НА ГРУБА СИЛА (BRUTE FORCE)

- **Задача:** На колку начини може да се постават две кралици на шаховска табла (8x8) без да се напаѓаат. Две кралици се напаѓаат ако се во иста редица или иста колона или на иста дијагонала.
- **Решение**: Ќе ги најдеме сите можни позиции на кои можат да се најдат кралиците и ќе ги изброиме само оние позиции на кои не се напаѓаат.

ТЕХНИКА НА ГРУБА СИЛА (BRUTE FORCE)

```
public static int seNapagjaat(int x1, int y1, int x2, int y2) {
return (x1 == x2 || y1==y2  || (abs(x1-x2)==abs(y1-y2)) ) ;  } //(5,6) – (1,2)
public static int nachini() {
int x1,y1,x2,y2;
int broj= 0;
for (x1=0;x1<8;x1++)
 for (y1=0;y1<8;y1++)
        for (x2=0;x2<8;x2++)
              for (y2=0;y2<8;y2++)
              if (!seNapagjaat(x1, y1, x2, y2) )
              broj++;
return broj;   }
```

ТЕХНИКА НА ГРУБА СИЛА (BRUTE FORCE)

- **Задача:** Даден е неограничен број на парички со вредност 50, 20, 10, 2, 1 денари. За дадена сума колкав е минималниот број на парички кои ја формираат таа сума?
- **Решение:** Со 5 for циклуси ќе ги изминеме сите можни комбинации на парички и ќе ги разгледуваме само оние кои го даваат бараниот збир. Од нив ќе го бараме минималниот број на парички.

ТЕХНИКА НА ГРУБА СИЛА (BRUTE FORCE)

```
public static int min_parichki (int suma) {
final static MAX=100;
int p1,p2,p3,p4,p5,pom, min= MAX, brParichki;
for (p1=0;p1<=(suma/50);p1++)
for (p2=0;p2<=(suma/20);p2++)
for (p3=0;p3<=(suma/10);p3++)
for (p4=0;p4<=(suma/2);p4++)
for (p5=0;p5<=(suma);p5++)
{pom = p1*50+p2*20+p3*10+p4*2+p5;
if (pom == suma) {
brParichki = p1+p2+p3+p4+p5;
if (brParicki < min)
min= br_paricki;
}   }
return min;    }
```

Комплексност =>
O(suma*(suma/2)*(suma/10)*(suma/20)*(suma/50))
Како да се забрза алгоритмот? => употребете алчна стратегија

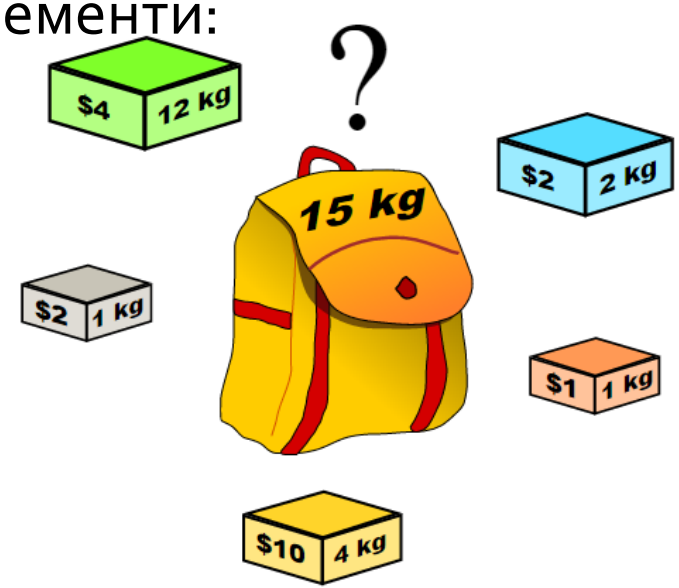
ТЕХНИКА НА ГРУБА СИЛА (BRUTE FORCE)

- **Задача (пакување на ранец – knapsack problem):** Дадени се n елементи кои треба да се спакуваат

во ранец со капацитет C. Секој елемент има тежина (t): t_1 t_2 … t_n и вредност (v): v_1 v_2 … v_n. Најдете ги елементите со најголема вредност кои можат да се сместат во ранецот.

Пример: Нека е даден ранец со капацитет C=16 и следниве елементи:

елемент	тежина	вредност
1	2	$20
2	5	$30
3	10	$50
4	5	$10



ТЕХНИКА НА ГРУБА СИЛА (BRUTE FORCE)

- Треба да се изминат следниве комбинации и да се одбере она решение кое не го надминува капацитетот на ранецот, а има најголема вредност

Подмножество	Вкупна тежина	Вкупна вредност
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	ја надминува тежината на ранецот
{1,2,4}	12	$60
{1,3,4}	17	ја надминува тежината на ранецот
{2,3,4}	20	ја надминува тежината на ранецот
{1,2,3,4}	22	ја надминува тежината на ранецот

елемент	тежина	вредност
1	2	$20
2	5	$30
3	10	$50
4	5	$10

За n предмети бројот на комбинации е 2^n
Комплексност => $O(2^n)$
Како да се забрза алгоритмот? => употребете алчна стратегија

АЛЧНИ АЛГОРИТМИ (GREEDY)

- Се прават одлуки инкрементално во мали чекори, без да се враќаме наназад.
- Го креираме решението со додавање на елементи, еден по еден, користејќи едноставно правило.
- Секогаш се бара локалното оптимално решение
- Одлуката во секој чекор треба да ја оптимизира тековната состојба, без да се води сметка дали таа одлука е најдобра и во иднина
- Одлуките се донесуваат според фиксни и едноставни правила на приоритет
  - (+) Едноставни за дизајнирање
  - (+) Едноставни за имплементирање и прилично брзи
  - (-) **Не** даваат секогаш најоптимално и најпрецизно решение
- Се применуваат кај проблеми кои користат распоредување

АЛЧНИ АЛГОРИТМИ (GREEDY)

- Алчните алгоритми се применуваат на посебен тип проблеми. Треба да правиме избори. Во секој чекор имаме множество од можни избори. Кога алгоритмот ќе заврши, добиено е конечно множество од избори, **c[1], c[2], c[3], …, c[n]**. Претпоставуваме дека со секој избор е поврзана некаква цена. Да претпоставиме дека наша цел е да го најдеме коректното решение со најмала цена.
- Исто така претпоставуваме дека при правењето на изборот **c[i]**, остатокот од проблемот е од ист тип.
- Алчниот алгоритам прави избори следејќи многу едноставна (алчна) стратегија. Стратегијата зависи од конкретниот проблем. Најчесто има два типа на алчни стратегии:
  - Го правиме изборот **c[i]**, т.ш. цената (локално) се зголемува што е можно помалку (ова е најчеста стратегија)
  - Го правиме изборот **c[i]**, т.ш. остатокот од проблемот е што е можно подобар

АЛЧНИ АЛГОРИТМИ (GREEDY)

- Алчниот алгоритам се извршува на следниов начин: претпоставуваме дека веќе сме ги направиле изборите c[1], c[2], …, c[m]. Ако ваквиот избор е точен, завршуваме. Поинаку, следниот чекор треба да ја следи претходната стратегија.
- Главната идеја на алчните алгоритми е тоа што не треба да трошиме време на правење на избори. Не треба да гледаме нанапред и да ги разгледуваме последиците од тековно направениот избор. Заради тоа алчните алгоритми имаат мала комплексност.

АЛЧНИ АЛГОРИТМИ (GREEDY)

- Алчните алгоритми може да не успеат заради две причини:
  - Не даваат оптимален резултат
    - Ако ги имаме на располагање броевите 6,5,1 и треба да најдеме збир N=10, алгоритмот ќе врати a=1, b=0, c=4, односно 10=6+1+1+1+1, а оптималното решение е 10=5+5
  - Не даваат точен резултат
    - Ако ги имаме на располагање броевите 6,5,2 и треба да најдеме збир N=7, алгоритмот ќе врати a=1, b=0, c=0, односно збирот 6=N, а оптималното решение е 7=5+2

АЛЧНИ АЛГОРИТМИ (GREEDY)

- **Задача:** Дадени се парички со вредност 20, 10, 50, 1, 2 денари. За дадена сума колкав е минималниот број на парички кои ја формираат таа сума?
- **Решение:** Најпрво треба да одлучиме како да ја сортираме влезната низа од парички. Ако ја сортираме во опаѓачки редослед, во секој чекор ќе ја земаме најголемата вредност на паричка и се додека може ќе ја одземаме од сумата. Тоа ќе го правиме и со останатите парички се дури сумата не стане нула или додека не ја изминеме цела низа со парички (ако сумата не стане нула)

АЛЧНИ АЛГОРИТМИ (GREEDY)

```
public static int min_par_gr (int [] par, int suma) //suma=120
{
 int i,j,  br=0, pom;
 int [ ] brpar=new int[5];
 for(i=0; i<par.length -1; i++)
 for(j=i+1; j<par.length; j++)
 if(par[i]<par[j]) {pom=par[i]; par[i]=par[j]; par[j]=pom;}
 i=0;
 while(suma>0 && i<par.length)
 {
  brpar[i]=suma/par[i]; //120/50 = 2 ; 20/20
  suma-=brpar[i]*par[i]; //120-2*50=20 //1*20;suma=0(ostatok)
  br+=brpar[i]; i++;  //br = br +2 +1 br na pari
 }
 return br;
 }
```

АЛЧНИ АЛГОРИТМИ (GREEDY)

- **Задача (пакување на ранец):** Дадени се n елементи x1,x2,…,xn кои треба да се спакуваат во ранец со капацитет C. Секој елемент има тежина (t): t_1 , t_2 … t_n и вредност (v): v_1 v_2 … v_n. Најдете ги елементите со најголема вредност кои можат да се сместат во ранецот.

Цел: треба да ја максимизираме вкупната вредност на предмети кои може да се сместат во ранецот $\sum_{i=1}^{n} x_i v_i$, а притоа да не се надмине капацитетот на ранецот, $\sum_{i=1}^{n} x_i t_i \leq C$

- **Ограничување**: 0<=x_i<=1

АЛЧНИ АЛГОРИТМИ (GREEDY)

- Нека бројот на објекти, n=3, тежините се (t1,t2,t3)=(18,15,10), а вредностите се (v1,v2,v3)=(25,24,15). Нека се дадени следниве решенија (во првата колона е прикажано колкав дел од објектот се зема)

(x1,x2,x3)	Вкупна тежина:	Вкупна вредност:
(1/2,1/3,1/4)	½*18+1/3*15+1/4*10=16.5	24.25
(1,2/15,0)	20	28.2
(0,2/3,1)	20	31
(0,1,1/2)	20	31.5

АЛЧНИ АЛГОРИТМИ (GREEDY)

- Оптимално решение се добива ако капацитетот на ранецот е поголем или еднаков на збирот на тежините на сите објекти. Тогаш сите објекти ќе може да се сместат во ранецот.
- Како да се одбере кој објект да се смести во ранецот:
  - **Сортирање (алчност) според тежина –** елементот со најмала тежина се проверува дали може да се смести цел (решение 3)
  - **Сортирање (алчност) според вредност (профит) –** во ранецот се сместува објект со следна најголема вредност. Ако не го собира цел, се вклучува само дел од него (решение 2)
  - **Сортирање (алчност) според однос вредност/тежина –** баланс меѓу брзината на зголемување на вредноста и брзината на искористување на капацитетот. Во секој чекор треба да се вклучува објект со максимална вредност по единица искористен капацитет, односно објектите треба да се подредени по однос вредност/тежина (решение 4)

(1/2,1/3,1/4)	½*18+1/3*15+1/4*10=16.5	24.25
(1,2/15,0)	20	28.2
(0,2/3,1)	20	31
(0,1,1/2)	20	31.5

АЛЧНИ АЛГОРИТМИ (GREEDY)

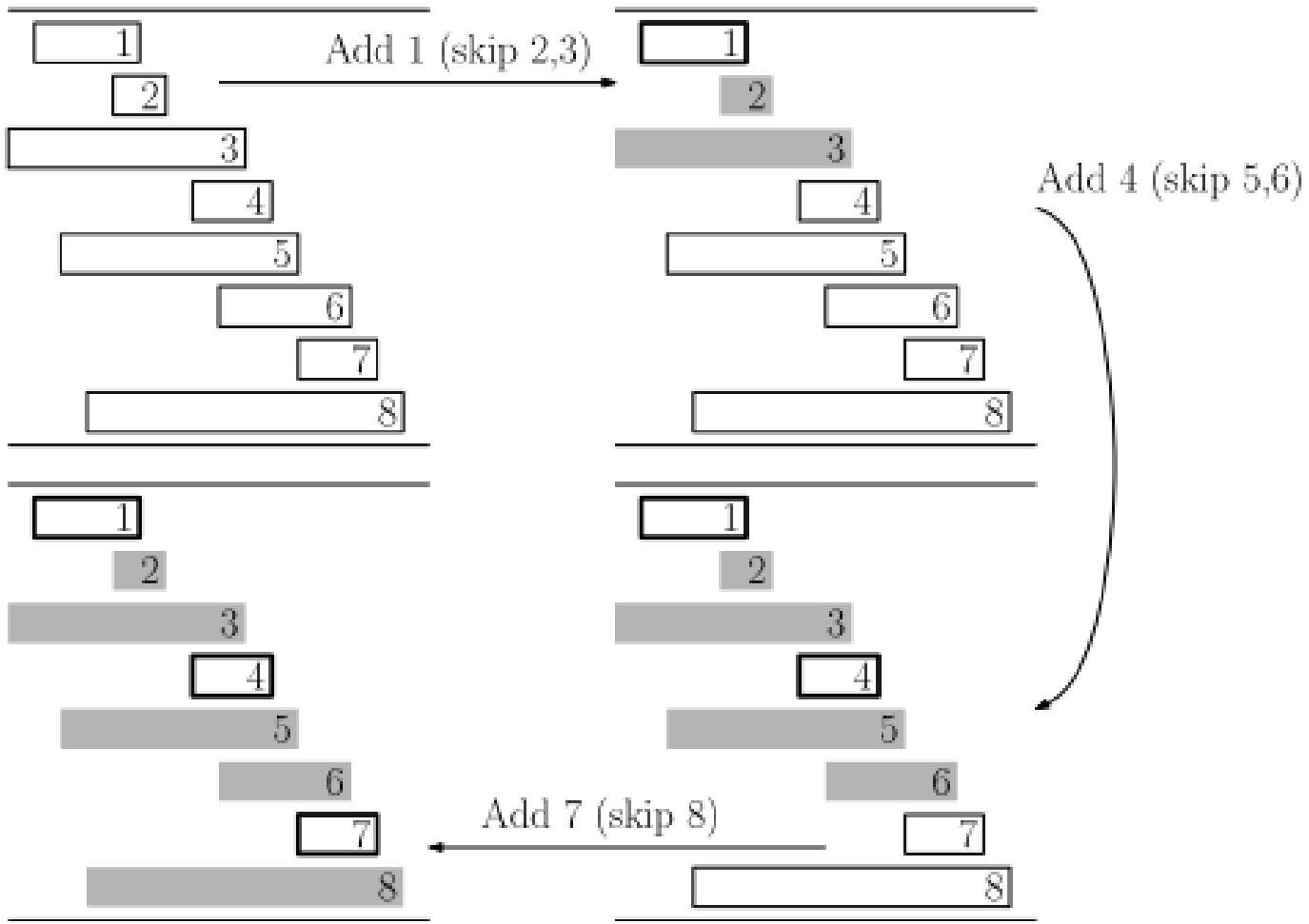
- **Задача (пакување на ранец):**

```
public static void fractionalKnapsack(int [ ] v, int [ ] t, int C, [ ] int x)
{ //pretpostavuvame deka nizite v i t se vekje sortirani spored odnosot vrednost/tezhina vo opagjachki redosled
 int i, n=v.length;
 for(i=0; i<n; i++) x[i]=0; //vo nizata x[ ] gi smestuvame reshenijata na problemot
 i=0;
 while(i<n && C>=t[i])
 { x[i]=1;//prviot objekt i site ostanati koi celosno gi sobira vo ranecot
//gi dodavame celi
 C-=t[i];
 i++;
 }
 if(i<n) x[i]=C/t[i]; //na kraj vo ranecot dodavame del od naredniot nevnesen //objekt
}
```

АЛЧНИ АЛГОРИТМИ (GREEDY)

- **Задача (временско распределување):** Дадено е множество од n активности A={a1,a2,…,an}. Секоја активност има почетно време, si, и крајно време, fi. Постои само еден опслужувач. Наша задача е да распределиме што повеќе активности на опслужувачот, но да внимаваме нивните временски интервали да не се преклопуваат.
- **Решение:** Постојат повеќе критериуми според кои можат да се распределуваат активностите: според време на траење, според време на започнување на активност, според време на завршување на активност.
- Ние ќе ги распределуваме според времето на завршување, односно сите активности ќе ги сортираме во растечки редослед според времето на завршување, а потоа во сортираната низа инкрементално ќе бараме активности кои временски не се поклопуваат

АЛЧНИ АЛГОРИТМИ (GREEDY)



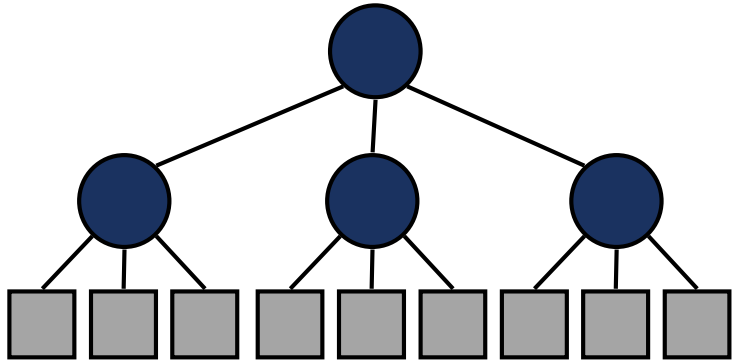
АЛЧНИ АЛГОРИТМИ (GREEDY)

- **Задача (временско распределување):**

```
public static void raspredeli(int [ ] a, int [ ] s, int [ ] f) //s – start time, f – finish time, a – activities
{       int i,j,k=1, n=a.length;
 for(i=0; i<n-1; i++)
        for(j=i+1; j<n; j++) //gi sortirame istovremeno trite nizi spored vreme  na zavrshuvanje na aktivnost
        if(f[i]>f[j]) {//kod za swap(f[i],f[j]); swap(s[i],s[j]); swap(a[i],a[j]);}
 int konechni[100]={0};
 konechni[0]=a[0]; //prvata aktivnost so najmalo vreme na zavrshuvanje
 j=0;
 for(i=1;i<n;i++)
 if(s[i]>f[j]) { konechni[k++]=a[i]; j=i; } //ako slednata aktivnost zapochnuva otkako kje zavrshi prethodnata,
// ja stavame vo nizata I ja postavuvame kako prethodna
}
```

РАЗДЕЛИ И ВЛАДЕЈ (DIVIDE AND CONQUER)

- **Чекори:**
  - **Раздели** – раздели го влезното множество S во две или повеќе дисјунктни множества S1, S2,… Рекурзивно решавај ги потпроблемите се додека не дојдеш до основен случај
  - **Владеј** – комбинирај ги решенијата за S1, S2, … во единствено решение за S
- Основниот случај за рекурзија се потпроблеми со константна големина



ЗАДАЧА

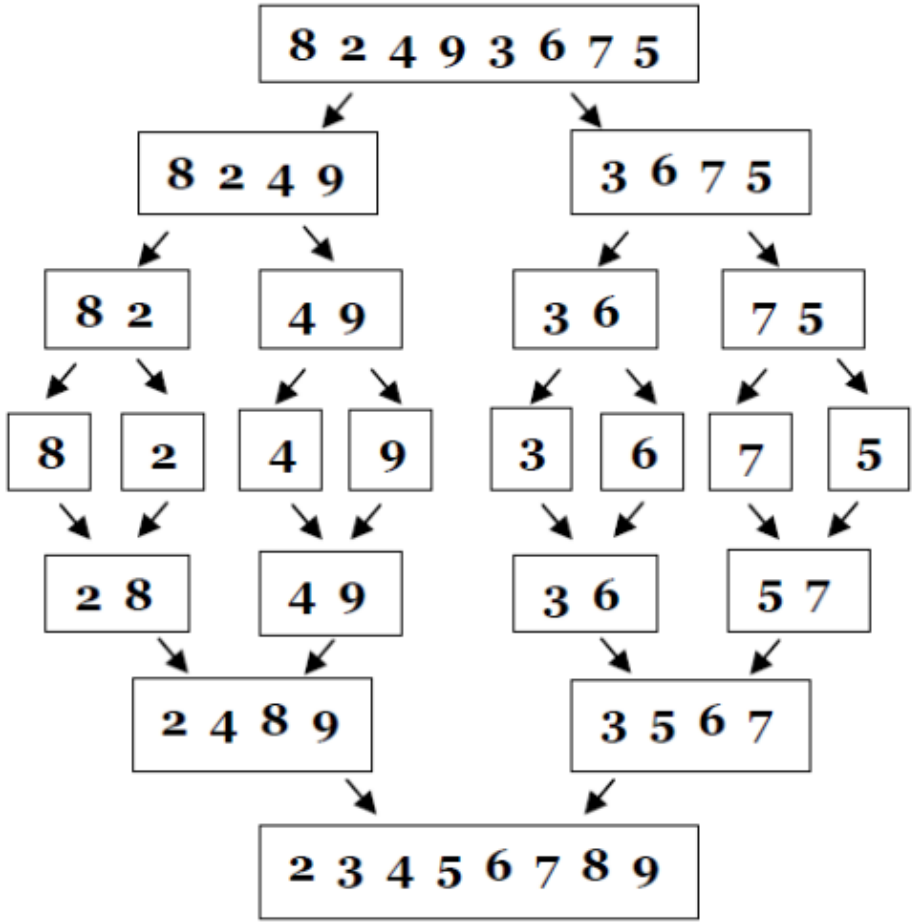
- **Задача:** Да се најде максималниот елемент во дадена (несортирана) низа од цели броеви. Низата да не се сортира
- **Решение:** Алгоритмот ја дели низата на a[0..n] на две низи a[0..m] a[m+1..n], го наоѓа максимумот на двате низи (рекурзивно), и го враќа поголемиот од тие два максимуми како максимум на целата низа. Основен случај е кога низата има еден елемент, тој елемент е максимумот.
- Ако големината на низата е парен број, таа е поделена на два еднакви дела, ако е непарен големините на двете низи се разликуваат за 1.

```
int max(int  [ ] a, int l, int r){
   if (l == r) return a[l];  //основен случај
   int m = (l+r)/2;
   int u = max(a, l, m);  //раздели
   int v = max(a, m+1, r);
   if (u > v) return u;  //владеј (состави)
  else return v;
 }
```

ЗАДАЧА - MERGE SORT

- Да се сортира дадена низа со помош на алгоритмот **merge sort**, односно сортирање со спојување
  - Раздели: подели ја низата на половина, во две дисјунктни поднизи
  - Владеј: секоја подниза сортирај ја рекурзивно
  - Состави: сортираните поднизи од секој рекурзивен повик, состави ги во една сортирана низа

ЗАДАЧА - MERGE SORT



ЗАДАЧА - MERGE SORT

```
void mergesort(int  [ ] a, int l, int r) {
if (l == r) {
return;
}
int mid = (l + r) / 2;
mergesort(a, l, mid);
mergesort(a, mid + 1, r);
merge(a, l, mid, r);
}
}
```

ЗАДАЧА - MERGE SORT

```
void merge(int [ ] a, int l, int mid, int r) {
int brel = r - l + 1;
int pom[] = new int[100]; // vo ovaa niza kje gi smestuvame sortiranite elementi
int i = l, j = mid+1, k = 0;
while ((i <= mid) && (j <= r)) {
if (a[i] < a[j]) {
pom[k] = a[i];
i++;
} else {
pom[k] = a[j];
j++;
}
k++;
}
```

ЗАДАЧА - MERGE SORT

```
while (i <= mid) {
pom[k] = a[i];
i++;
k++;
}
while (j <= r) {
pom[k] = a[j];
j++;
k++;
}
for (k = 0; k < brel; k++) {  //elementite mora da gi vratime vo originalnata niza so koja ja povikavme funkcijata
a[l + k] = temp[k];
}
```

ДИНАМИЧКО ПРОГРАМИРАЊЕ

- Техника која го дели проблемот на повеќе потпроблеми и потоа го пресметува секој потпроблем, така што неговото решение е оптимално
- Динамичкото програмирање се применува кога потпроблемите се зависни еден од друг (за разлика од раздели и владеј), односно делат некакви информации

ДИНАМИЧКО ПРОГРАМИРАЊЕ

- Алгоритмите за динамичко програмирање го решаваат секој потпроблем само еднаш и неговиот резултат го чуваат во табела. На овој начин се избегнуваат повторни непотребни пресметки секој пат кога ќе наидеме на истиот потпроблем.
- Динамичкото програмирање најчесто се употребува при решавање на оптимизациски проблеми.
- Во оптимизациските проблеми имаме многу можни решенија. Секое решение има своја вредност, но сакаме да го најдеме она решение со оптимална (минимална или максимална) вредност. Ваквото решение го викаме оптимално решение на проблемот.

ДИНАМИЧКО ПРОГРАМИРАЊЕ

- Пишувањето на динамички алгоритам може да се подели во 4 чекори:
  - 1. Наоѓање на структура на оптималното решение.
  - 2. Рекурзивно дефинирање на вредноста на оптималното решение.
  - 3. Пресметка на вредноста на оптималното решение со bottom-up пристап.
  - 4. Конструирање на оптимално решение од пресметаната информација.
  - Чекорите 1-3 ја формираат основата на динамичкото програмирање.
  - Чекорот 4 може да се изостави доколку се бара само вредноста на оптималното решение

ДИНАМИЧКО ПРОГРАМИРАЊЕ

- Елементи на динамичкото програмирање:
  - Оптимална подструктура
  - Потпроблеми кои се преклопуваат.

ЗАДАЧА – МАКСИМАЛЕН ЗБИР ВО МАТРИЦА

- Во секое поле од mxn матрица се внесени цели броеви. Дозволено ни е да се движиме само во десно и надолу.
- Како да го најдеме максималниот збир на елементи од даден почеток до даден крај во матрицата
- Решение:
  - Best(m,n)=max(Best(m,n-1), Best(m-1,n))+A(m,n)
  - Почетни состојби (познати): пресметка на максималните збирови доколку се движиме само во прва редица и пресметка на максимални збирови доколку се движиме само во прва колона.

    Best(0,0) = A(0,0)

ЗАДАЧА – МАКСИМАЛЕН ЗБИР ВО МАТРИЦА

Поч. 1	2	5	8	9	7
6	7	2	5	5	1
2	5	3	6	6	3
2	4	4	2	3	5
8	9	3	1	2	**Крај 3**

A[i][j]

Поч. 1	3	8	16	25	32
7					
9					
11					
19					**Крај ?**

B[i][j] – почетни вредности

Поч. 1	3	8	16	25	32
7	14	16			
9	19	22			
11					
19					**Крај ?**

ЗАДАЧА – МАКСИМАЛЕН ЗБИР ВО МАТРИЦА

```
int maksimalenZbir(int [ ] [ ] A, int m, int n) {
int [ ] [ ] B = new int [m] [n];
// pochetni sostojbi
B[0][0] = A[0][0];
for (int i = 1; i < m; i++) // ja popolnuvame prvata kolona
B[i][0] = B[i - 1][0] + A[i][0];
for (int i = 1; i < n; i++)
B[0][i] = B[0][i - 1] + A[0][i]; // ja popolnuvame prvata redica
for (int i = 1; i < m; i++)
for (int j = 1; j < n; j++)
B[i][j] = Math.max(B[i-1][j], B[i][j-1])+A[i][j]; //maksimalno od reshenijata vo gornoto pole i poleto levo od dadenoto
return B[m-1][n-1];
}
```