o cílových architekturách. Pokud je dobře navržen popis architektury, lze také často sdílet některé části back-endů, a některé optimalizace se tak přesouvají spíše do middle endu.

Úloha:

Abychom si pouze nepovídali, zkusíme si v této sérii prakticky napsat jednoduchý front-end k překladači. Abychom se vyhnuli komplikacím, měl by umět pouze překládat výrazy popsané gramatikou uvedenou v textu seriálu, do popsaného mezikódu. Výsledek výrazu by měl být přiřazen do proměnné `result`. Například výraz

```
x * (x + y) - z / bla / neco * 5
```

by měl být přeložen jako

```
assign tmp1 (x + y)
assign tmp2 (x * tmp1)
assign tmp3 (z / bla)
assign tmp4 (tmp3 / neco)
assign tmp5 (tmp4 * 5)
assign result (tmp2 - tmp5)
```

Recepty z programátorské kuchařky

I letos Vám kromě úloh budeme servírovat také recepty z programátorské kuchařky. Některé si vypůjčíme z dřívějších ročníků KSP, ale i k těm se budeme snažit připsat něco nového, aby si i zkušenější řešitelé přišli na své. V letošní první kuchařce si povíme o třídících algoritmech. Co to znamená? Pojem *třídění* je možná maličko nepřesný, nehodláme data (čísla, záznamy, řetězce a jiné) rozdělovat do nějakých tříd, ale přerovnat je do správného pořadí, protože se seřazenými údaji se mnohem lépe pracuje, například pokud v nich pak potřebujeme vyhledávat. Takové uspořádávání dat je denním chlebem každého programátora, a tak není divu, že třídící algoritmy jsou jedny z nejstudovanějších. My však nebudeme do nějakých velkých detailů a specialit příliš zabíhat. Zkrátka a dobře – budeme chtít třídit údaje rychle, úsporně a radostně.

Obvykle třídíme exempláře datové struktury typu pascalského záznamu. V takové datové struktuře bývá obsažena jedna význačná položka, *klíč*, podle které se záznamy řadí. Malinko si náš život zjednodušíme a budeme předpokládat, že třídíme záznamy obsahující pouze klíč, který je navíc celočíselný – budeme tedy třídit pole celých čísel. Pomocí počtu tříděných čísel N pak budeme vyjadřovat časovou (a paměťovou) složitost jednotlivých algoritmů, které si předvedeme.

Metody třídění můžeme rozdělit do dvou hlavních skupin, a to na *vnitřní třídění,* kdy si můžeme dovolit všechna data načíst do (rychlé) paměti počítače, a na *vnější třídění,* kdy již třídění musíme realizovat opakovaným čtením a vytvářením diskových souborů. V tomto dílu se omezíme pouze na algoritmy vnitřního třídění a tříděné pole si nadeklarujeme takto:

```
const N = 100;
type Pole = array[1..N] of integer;
```

Nejjednodušší třídící algoritmy patří do skupiny *přímých metod.* Všechny mají několik společných rysů: Jsou krátké, jednoduché a třídí přímo v poli (nepotřebujeme pomocné pole). Tyto algoritmy mají většinou časovou složitost $\mathcal{O}(N^2)$. Z toho vyplývá, že jsou použitelné tehdy, když tříděných dat není příliš mnoho. Na druhou stranu pokud je dat opravdu málo, je zbytečně složité používat některý z komplikovanějších algoritmů, které si předvedeme později.

Stručně si přiblížíme tři nejznámější algoritmy pro třídění přímými metodami. *Třídění přímým výběrem (SelectSort)* je založeno na opakovaném vybírání nejmenšího čísla z dosud nesetříděných čísel. Nalezené číslo prohodíme s prvkem na začátku pole a postup opakujeme, tentokrát s nejmenším číslem na indexech $2, \ldots, N$, které prohodíme s druhým prvkem v poli. Poté postup opakujeme s prvky s indexy $3, \ldots, N$, atd. Je snadné si uvědomit, že když takto postupně vybíráme minimum z menších a menších intervalů, setřídíme celé pole (v i-tém kroku nalezneme i-tý nejmenší prvek a zařadíme ho v poli na pozici s indexem i).

```
procedure SelectSort(var A: Pole);
var i,j,k,x: integer;
begin
  for i:=1 to N-1 do
    begin
      k:=i;
      for j:=i+1 to N do
        if A[j]<A[k] then k:=j;
      x:=A[k]; A[k]:=A[i]; A[i]:=x;
    end;
end;
```

Pro úplnost si ještě řekněme pár slov o časové složitosti právě popsaného algoritmu. V i-tém kroku musíme nalézt minimum z $N-i+1$ čísel, na což spotřebujeme čas $\mathcal{O}(N-i+1)$. Ve všech krocích dohromady tedy spotřebujeme čas $\mathcal{O}(N+(N-1)+\ldots+3+2+1) = \mathcal{O}(N^2)$.

Třídění přímým vkládáním (InsertSort) funguje na podobném principu jako třídění přímým výběrem. Na začátku pole vytváříme správně utříděnou posloupnost, kterou postupně rozšiřujeme. Na začátku i-tého kroku má tato utříděná posloupnost délku $i-1$. V i-tém kroku určíme pozici i-tého čísla v dosud utříděné posloupnosti a zařadíme ho do utříděné posloupnosti (zbytek utříděné posloupnosti se posune o jednu pozici doprava). Není těžké si rozmyslet, že každý krok lze provést v čase $\mathcal{O}(N)$. Protože počet kroků algoritmu je N, celková časová složitost právě popsaného algoritmu je opět $\mathcal{O}(N^2)$.

```
procedure InsertSort(var A: Pole);
var i,j,x: integer;
begin
  for i:=2 to N do
    begin
      x:=A[i];
      j:=i-1;
      while (j>0) and (x<A[j]) do
        begin
          A[j+1]:=A[j];
          j:=j-1;
        end;
      A[j+1]:=x;
    end;
end;
```

(Upozornění: v našich příkladech předpokládáme, že máme v překladači zapnuto tzv. zkrácené vyhodnocování logických výrazu, třeba v předchozím while-cyklu se při `j=0` hodnoty `x` a `A[0]` již neporovnávají.)

Bublinkové třídění (BubbleSort) pracuje jinak než dva dříve popsané algoritmy. Algoritmu se říká „bublinkový“, protože podobně jako bublinky v limonádě „stoupají“ vysoká čísla v poli vzhůru. Postupně se porovnávají dvojice sousedních prvků, řekněme zleva doprava, a pokud v porovnávané dvojici následuje menší číslo po větším, tak se tato dvě čísla prohodí. Celý postup opakujeme, dokud probíhají nějaké výměny. Protože algoritmus skončí, když nedojde k žádné výměně, je pole na konci algoritmu setříděné.

```
procedure BubbleSort(var A: Pole);
var i,x: integer;
    zmena: boolean;
begin
  repeat
    zmena:=false;
    for i:=1 to N-1 do
      if A[i] > A[i+1] then
        begin
          x:=A[i]; A[i]:=A[i+1]; A[i+1]:=x;
          zmena:=true;
        end;
  until not zmena;
end;
```

Správnost algoritmu nahlédneme tak, že si uvědomíme, že po i průchodech while-cyklem bude posledních i prvků obsahovat největších i prvků setříděných od nejmenšího po největší (rozmyslete si, proč tomu tak je). Popsaný algoritmus se tedy zastaví po nejvýše N průchodech a jeho celková časová složitost v nejhorším případě je $\mathcal{O}(N^2)$, neboť na každý průchod spotřebuje čas $\mathcal{O}(N)$. Výhodou tohoto algoritmu oproti předchozím dvěma algoritmům je, že pokud je pole na začátku setříděné, tak algoritmus spotřebuje jen lineární čas, $\mathcal{O}(N)$.

Lepší třídící algoritmy pracují v čase $\mathcal{O}(N \log N)$. Jedním z nich je *Třídění sléváním (MergeSort),* založené na principu slévání (spojování) již setříděných posloupností dohromady. Představme si, že již máme dvě setříděné posloupnosti a chceme je spojit dohromady. Jednoduše stačí porovnávat nejmenší prvek z každé posloupnosti, který jsme dosud nedali do nově vytvářené posloupnosti, a menší z těchto prvků do nové posloupnosti přidat. Je zřejmé, že ke slití dvou posloupností potřebujeme čas úměrný součtu jejich délek.

My si zde popíšeme a předvedeme modifikaci algoritmu MergeSort, která používá pomocné pole. Algoritmus lze implementovat při zachování časové složitosti i bez pomocného pole, ale je to o dost pracnější. Existuje též modifikace algoritmu, která má počet fází (viz dále) v nejhorším případě $\mathcal{O}(\log N)$, ale pokud je již pole na začátku setříděné, proběhne pouze jediná a v takovém případě má algoritmus časovou složitost $\mathcal{O}(N)$. My si však zatajíme i tuto variantu.

Algoritmus pracuje v několika *fázích.* Na začátku první fáze tvoří každý prvek jednoprvkovou setříděnou posloupnost a obecně na začátku i-té fázi budou mít setříděné posloupnosti délky 2^{i-1}. V i-té fázi tedy vždy ze dvou sousedních 2^{i-1}-prvkových posloupností vytvoříme jedinou délky 2^i. Pokud N není násobkem 2^i, bude délka poslední posloupnosti zbytek po dělení N číslem 2^i. Zastavíme se, pokud $2^i \geq N$, tj. po $\lceil \log_2 N \rceil$ fázích. Protože v i-té fázi slijeme $\lceil N/2^i \rceil$ dvojic nejvýše 2^{i-1}-prvkových posloupností, je časová složitost jedné fáze $\mathcal{O}(N)$. Celková časová složitost popsaného algoritmu je pak $\mathcal{O}(N \log N)$.

```
procedure MergeSort(var A: Pole);
var P: Pole;        { pomocné pole }
    delka:integer;  { délka setříděných posl. }
    i: integer;     { index do vytvářené posl. }
    i1,i2: integer; { index do slévaných posl. }
    k1,k2: integer; { konce slévaných posl. }
begin
  delka:=1;
  while delka<N do
    begin
      i1:=1; i2:=delka+1; i:=1;
      k1:=delka; k2:=2*delka;
      while i<=N do
        begin
          if k2>N then k2:=N;
          while (i1<=k1) or (i2<=k2) do
            if (i1>k1) or
               ((i2<=k2) and (A[i1]<=A[i2]))
            then
              begin
                P[i]:=A[i1]; i:=i+1; i1:=i1+1;
              end
            else
              begin
                P[i]:=A[i2]; i:=i+1; i2:=i2+1;
              end;
          i1:=k2+1; i2:=k2+delka;
          k1:=k2+delka;
          k2:=k2+2*delka;
        end;
      A:=P;
      delka:=2*delka;
    end;
end;
```

V čase $\mathcal{O}(N \log N)$ pracuje také algoritmus jménem *QuickSort.* Tento algoritmus je založen na metodě Rozděl a panuj. Nejprve si zvolíme nějaké číslo, kterému budeme říkat *pivot.* Více si o jeho volbě povíme později. Poté pole přeuspořádáme a rozdělíme je na dvě části tak, že žádný prvek v první části nebude větší než pivot a žádný prvek v druhé části naopak menší. Prvky v obou částech pak setřídíme rekurzivním zavoláním téhož algoritmu. Musíme ale dát pozor, aby v každém kroku obě části byly neprázdné (a rekurze tedy byla konečná). Je zřejmé, že po skončení algoritmu bude pole setříděné.

Malá zrada spočívá ve volbě pivota. Pro naše účely by se hodilo, aby po přeházení prvků levá i pravá část pole byly přibližně stejně velké. Nejlepší volbou pivota by tedy byl *medián* tříděného úseku, tj. prvek takový, jenž by byl v setříděném poli přesně uprostřed. Přeuspořádání jistě zvládneme v lineárním čase a pokud by pivoty na všech úrovních byly mediány, pak by počet úrovní rekurze byl $\mathcal{O}(\log N)$ a celková časová složitost $\mathcal{O}(N \log N)$ (na každé úrovni rekurze je součet délek tříděných posloupnosti nejvýše N). Ačkoli existuje algoritmus, který medián pole nalezne v čase $\mathcal{O}(N)$, v QuickSortu se obvykle nepoužívá, jelikož konstanta u členu N je příliš velká v porovnání s pravděpodobností, že náhodná volba pivota algoritmus příliš zpomalí. Většinou se pivot volí náhodně z dosud nesetříděného úseku – zkrátka se sáhne někam do pole a nalezený prvek se prohlásí za pivot. Dá se ukázat, že takovýto algoritmus s velmi vysokou pravděpodobností poběží v čase $\mathcal{O}(N \log N)$. Důkaz tohoto tvrzení je trošičku trikový a lze jej nalézt např. v knize Kapitoly z diskrétní matematiky od pánů Matouška a Nešetřila.