VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

APLIKACE NA PODPORU VÝUKY DYNAMICKÉHO PROGRAMOVÁNÍ

APPLICATION FOR THE DYNAMIC PROGRAMMING DEMONSTRATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **TOMÁŠ NEREČA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. IVANA BURGETOVÁ, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Nereča Tomáš**
Program: Informační technologie
Název: **Aplikace na podporu výuky dynamického programování**
Application for the Dynamic Programming Demonstration
Kategorie: Algoritmy a datové struktury

Zadání:

1. Seznamte se s technikou dynamického programování.
2. Prostudujte úlohy, které lze touto technikou efektivně řešit.
3. Po dohodě s vedoucí navrhněte aplikaci, která bude demonstrovat principy a výhody dynamického programování na vybraných úlohách.
4. Navrženou aplikaci implementujte.
5. Aplikaci otestujte a zhodnoťte dosažené výsledky.

Literatura:

- Mareš, M., Valla, T.: *Průvodce labyrintem algoritmů,* CZ.NIC, 2017, ISBN 978-80-88168-19-5.
- Cormen, T. H. *Introduction to algorithms*. 3rd ed. Cambridge: MIT Press, 2009, ISBN 978-0-262-03384-8.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Burgetová Ivana, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 30. října 2018

Abstrakt

Webová aplikácia, ktorá je výsledkom tejto práce, sa zaoberá technikou návrhu algoritmov s názvom dynamické programovanie. Aplikácia na konkrétnych príkladoch poukazuje na jej princípy a výhody. Pri každom príklade je konkrétny algoritmus teoreticky vysvetlený a jeho priebeh je znázornený pomocou dynamicky vypĺňanej tabuľky. Okrem toho aplikácia porovnáva efektivitu riešenia technikou dynamického programovania s jednoduchým rekurzívnym riešením pomocou grafov a tabuľky.

Abstract

A result of this bachelor thesis is a web application which focuses on computer programming method called dynamic programming. Principles and advantages of dynamic programming are explained on several examples. Dynamic programming algorithm is explained both theoretically and also practically by a dynamically filled up table. Dynamic programming solution is also compared with simple recursive solution in charts and table.

Kľúčové slová

DP, dynamické programovanie, rekurzívny algoritmus, programovacia technika, optimalizačná úloha, React, TypeScript, Material-UI

Keywords

DP, dynamic programming, recursive algorithm, computer programming method, optimization problem, React, TypeScript, Material-UI

Citácia

NEREČA, Tomáš. *Aplikace na podporu výuky dynamického programování*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Ivana Burgetová, Ph.D.

Aplikace na podporu výuky dynamického programování

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Ing. Ivany Burgetovej Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Tomáš Nereča
15. mája 2019

Poďakovanie

Rád by som poďakoval Ing. Ivane Burgetovej Ph.D. za cenné rady, pripomienky a trpezlivosť pri vedení mojej bakalárskej práce.

Obsah

Kapitola 1

Úvod

Bežný človek musí dennodenne riešiť rôzne úlohy. Niektoré sú jednoduchšie, ako napríklad vyniesť odpadky, niektoré naopak zložitejšie, hlavne z pracovného prostredia. Cieľom každého z nás by malo byť efektívne riešenie týchto úloh. K tomu je potrebné si vopred navrhnúť vhodný postup riešenia, teda algoritmus. Poznáme viacero techník návrhu algoritmov a záleží od konkrétnej úlohy, ktorú z nich by sme mali zvoliť, aby vznikol čo najefektívnejší algoritmus. Práve dynamické programovanie je jednou z týchto techník.

Webová aplikácia **DP learn**, ktorá je výsledkom tejto práce, vysvetľuje túto techniku na sérii príkladov. Aplikácia je dostupná na adrese `https://dp-learn.firebaseapp.com/`. Užívateľ aplikácie si môže sám zvoliť vstupné hodnoty úlohy a sledovať priebeh algoritmu v dynamicky vypĺňanej tabuľke. Okrem toho aplikácia poskytuje aj teóriu k jednotlivým príkladom a vykresľuje aj grafy a tabuľky pre porovnanie efektivity jednoduchého rekurzívneho algoritmu a algoritmu navrhnutého technikou dynamického programovania. Všetko sa nachádza na jednom mieste. Ide teda o komplexné riešenie a práve týmto sa líši od voľne dostupných webových stránok a aplikácií, ktoré sa väčšinou špecializujú len na teóriu alebo grafické znázornenie priebehu algoritmu.

Aplikácia má slúžiť ako podporný materiál pri výučbe. Využitie teda nájde najmä v akademickej sfére, ale zaujímavá môže byť pre každého, kto sa aspoň trochu zaujíma o programovanie, algoritmizáciu alebo matematické výpočty.

Obsah tejto správy je rozdelený do niekoľkých kapitol. Tá nasledujúca 2 obsahuje nevyhnutnú teóriu k pochopenie princípov dynamického programovania. Bude vysvetlené, kedy je možné túto techniku použiť, aký je postup návrhu, a aké výhody prináša oproti rekurzívnemu riešeniu. Ďalšia kapitola 3 sa sústredí na návrh aplikácie. To zahŕňa zhodnotenie aktuálneho stavu a s tým súvisiace požiadavky na novú aplikáciu. Nasleduje návrh užívateľského rozhrania a výber vhodných technológií, na ktorých bude aplikácia postavená. Implementácia je popísaná v samostatnej kapitole 4. Po nej nasleduje kapitola 5, ktorá sa zaoberá testovaním aplikácie a experimentami prevedenými na optimalizačných úlohách. Záverečná kapitola 6 obsahuje zhrnutie výsledkov práce a popisuje možnosti na ďalšie vylepšenia a rozšírenia aplikácie.

Kapitola 2

Dynamické programovanie

Táto kapitola sa zaoberá dynamickým programovaním po teoretickej stránke. Bude vysvetlené, čo to vlastne dynamické programovanie je, a ako vznikol tento názov. Nasledovať bude priblíženie hlavných princípov, a teda aj výhody a nevýhody oproti jednoduchému rekurzívnemu algoritmu. V ďalšej časti budú prebrané vlastnosti, ktoré musí úloha spĺňať, aby malo zmysel navrhnúť algoritmus podľa princípov dynamického programovania. Následne budú jednotlivé kroky návrhu algoritmu vysvetlené na konkrétnom príklade. Na záver budú preberané ďalšie optimalizačné úlohy, ktoré sú súčasťou aplikácie.

2.1 Dynamické programovanie

Dynamické programovanie (skrátene **DP**[1]) je optimalizačná metóda, ktorá sa dá využiť pri riešení určitého typu *optimalizačných úloh*. Optimalizačná úloha je taká úloha, ktorá má viacero správnych riešení. Každé z týchto riešení má určitú hodnotu. Cieľom je nájsť extrém (najnižšiu alebo najvyššiu hodnotu). Riešení, ktorých výsledkom je extrém môže byť viacero. Môže teda existovať viacero optimálnych riešení [6]. Okrem toho, že je to optimalizačná metóda je dynamické programovanie aj programovacia technika – jeden z prístupov návrhu algoritmov.

Ako vlastne vznikol názov dynamické programovanie? Názov tejto metódy je istým spôsobom záhadný. Neudáva jasné vysvetlenie o tom, čo je jeho princípom. Jej autor, Richard Ernest Bellman, udáva dôvody tohto pomenovania vo svojej autobiografii [2], z ktorej som čerpal v nasledujúcich riadkoch. Jeho prvou úlohou, keď nastúpil v 50.-tych rokoch minulého storočia do organizácie RAND[2], bolo vymyslieť názov pre viacstupňové rozhodovacie procesy, v ktorých voľba optimálneho kroku záleží na predchádzajúcich krokoch. Toto obdobie veľmi neprialo výskumu v oblasti matematiky. Organizácia RAND patrila pod U.S. Air Force[3] a Charles Erwin Wilson ako minister obrany (a tým aj veliteľ U.S. Air Force) doslova nenávidel slovo „*research*“ (v preklade „výskum“). Bellman preto nechcel použiť názov, ktorý by poukazoval na to že v RAND prebieha nejaký matematický výskum. Napadali mu slová ako „rozhodovanie“ alebo „plánovanie“, ale nakoniec zvíťazilo slovo „programovanie“. Ďalej chcel poukázať na to, že ide o niečo viacstupňové, meniace sa v čase. Slovo „dynamické“ sa navyše nedalo použiť v pejoratívnom význame, preto to bola ideálna možnosť. Spojením týchto slov teda vznikol názov „dynamické programovanie“.

[1] **DP** je často používaná skratka pre **D**ynamické **P**rogramovanie. Často bude používaná aj v tejto práci.
[2] https://www.rand.org/
[3] https://www.af.mil/

Myšlienky, ktoré viedli k vzniku názvu už môžu naznačovať niečo z princípov tejto metódy. DP využíva rozklad problému na podproblémy. Tie sú rozkladané na ďalšie podproblémy, až dokým nevzniknú tak jednoduché vstupy, ktoré sa dajú vyriešiť priamo. Z výsledkov podproblémov sa potom dá postupne vytvoriť riešenie pôvodného problému. Rovnaký princíp využíva aj metóda *Rozdeľuj a panuj*[4], ktorej sa DP veľmi podobá, ale pridáva dôležitý krok navyše – **ukladanie výsledkov podproblémov** [23]. Často sa pri dynamickom programovaní navrhuje algoritmus prístupom zdola-hore, ktorá navyše otáča smer výpočtu. Výsledky podproblémov sú ukladané do vopred vytvorenej dátovej štruktúry, konkrétne poľa. Pole je vypĺňané *dynamicky*, hodnoty na rovnakých indexoch sú často prepisované niekoľkokrát. Ak algoritmus narazí znovu na podproblém, ktorý už bol predtým vyriešený, použije sa uložený výsledok. Tým sa ušetrí čas, ktorý by bol potrebný na opakovaný výpočet. DP algoritmus teda môže byť pri určitom type úloh oveľa efektívnejší ako rekurzívny algoritmus.

2.2 Vlastnosti úlohy riešiteľnej dynamickým programovaním

Zo spomínaných princípov DP vyplývajú vlastnosti, ktoré musí úloha spĺňať, aby bolo možné, a zároveň efektívne navrhnúť DP algoritmus k jej vyriešeniu. Podrobne sú popísané napríklad v knihe *Introduction to Algorithms* [6]. Ide o tieto dve vlastnosti:

1. Optimálna subštruktúra
2. Prekrývanie podproblémov

Aby bolo vôbec možné riešiť optimalizačnú úlohu pomocou dynamického programovania, musí mať úloha tzv. **optimálnu subštruktúru**. V prvom rade sa musí dať hlavný problém, ktorý treba vyriešiť, rozdeliť na podproblémy. Každý z týchto podproblémov má vlastné optimálne riešenie. Ak sa spojením optimálnych riešení jednotlivých podproblémov dá získať optimálne riešenie pôvodného problému, dá sa konštatovať, že úloha má optimálnu subštruktúru.

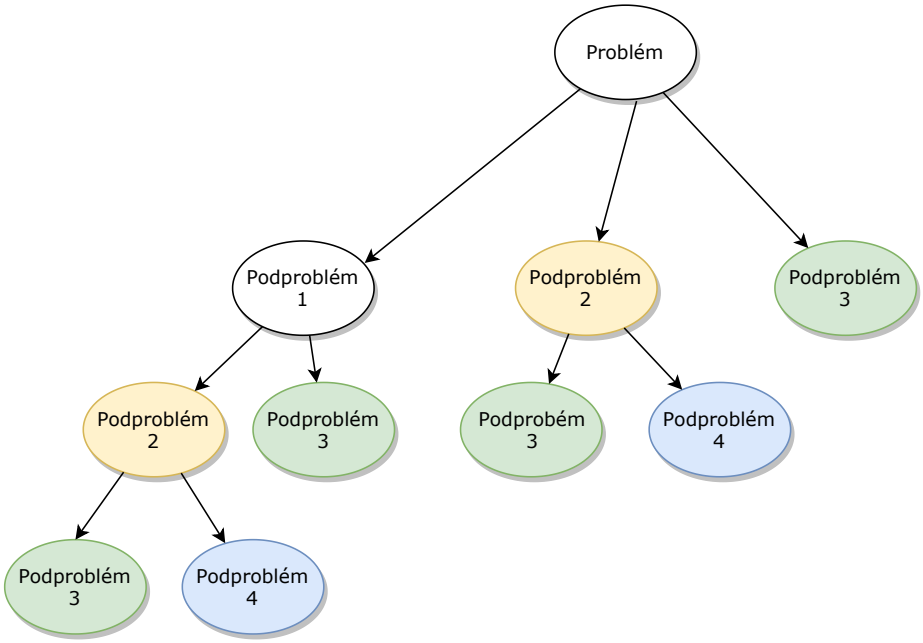
Prekrývanie podproblémov znamená, že sa v priebehu algoritmu viackrát opakuje výpočet pre rovnaké vstupné hodnoty, ktorý logicky dáva vždy rovnaký výsledok. Toto opakovanie je znamením, ktoré nabáda k využitiu dynamického programovania. Zistiť, či sa pri riešení úlohy vyskytujú podproblémy, ktorých výsledky sú vypočítavané opakovane je celkom jednoduché, ak poznáme rekurzívne riešenie. Stačí si zvoliť vstupné hodnoty úlohy a nakresliť strom rekurzívnych volaní (obrázok 2.1).

Pre pochopenie nasledujúcich riadkov je potrebné vysvetliť pojem *zložitosť algoritmu*. Tento pojem bol zavedený, aby bolo možné rozumne porovnávať efektivitu rôznych algoritmov. Rozlišujeme 2 druhy zložitosti:

- **Priestorová zložitosť** – udáva pamäťové požiadavky pri vykonávaní algoritmu
- **Časová zložitosť** – udáva počet operácií, ktoré algoritmus vykoná, pričom vykonanie každej takejto operácie zaberie určitý čas

Zložitosť sa obvykle zapisuje ako funkcia vstupných dát. Ide teda o matematické vyjadrenie, ktoré neudáva presnú hodnotu. Je to z toho dôvodu, že ten istý algoritmus môže bežať

[4] https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms



Obr. 2.1: Strom rekurzívnych volaní graficky znázorňuje priebeh rekurzívneho algoritmu. Pri konkrétnom zadaní by *Podproblém 3* bolo volanie funkcie s určitými parametrami. V strome na obrázku sa tento podproblém opakuje dokopy 4-krát. Funkcia by teda bola volaná 4-krát s rovnakými parametrami. Navyše, podproblém môže mať ďalšie podproblémy, ktoré sú vždy rovnaké a aj tieto budú vypočítavané opakovane.

rôzne dlho v závislosti od konkrétnej implementácie v konkrétnom jazyku, na konkrétnom stroji [21]. Pre vyjadrenie zložitosti sa používa niekoľko typov notácií. V tejto správe a tiež v aplikácii bude využívaná tzv. „*Big-Omicron*“ (v preklade „Veľké-Omicron“) notácia, ktorá má tvar $O(f(x))$. Táto notácia popisuje tzv. *asymptotické* chovanie funkcie, čo znamená chovanie v extrémnych prípadoch. Ide teda o akúsi maximálnu teoretickú hodnotu, ktorá môže nastať v závislosti od vstupných dát [20]. Na základe funkcie sa dá algoritmus zaradiť do *triedy zložitosti*. Triedy zložitosti sú napríklad **polynomická** (napr. $O(N^2)$) alebo **exponenciálna** (napr. $O(N^K)$ [25].

Vráťme sa teraz k stromu rekurzívnych volaní. Ak je zrejmé, že podproblémy sa neopakujú, navrhovať DP algoritmus nemá zmysel. Práve naopak by sa iba zvýšila priestorová zložitosť, pretože na ukladanie výsledkov podproblémov potrebujeme pamäťový priestor navyše. Cieľom DP algoritmu je znížiť triedu časovej zložitosti, ktorá býva pri rekurzívnom riešení často exponenciálna.

Ak sa potvrdí, že optimalizačná úloha má optimálnu subštruktúru, a zároveň sa opakuje riešenie rovnakých podproblémov, môžeme navrhnúť algoritmus pomocou techniky dynamického programovania.

2.3 Návrh algoritmu pomocou techniky dynamického programovania

Vytvorenie efektívneho algoritmu postaveného na princípoch dynamického programovania zahŕňa obvykle 4 kroky:

1. Charakterizovanie štruktúry optimálneho riešenia
2. Definovanie hodnoty optimálneho riešenia pomocou rekurzie
3. Výpočet hodnoty optimálneho riešenia prístupom zdola-hore
4. Zostavenie celého optimálneho riešenia úlohy

Prvé tri kroky je potrebné vykonať vždy. 4. krok je potrebný v prípade, ak chceme okrem hodnoty optimálneho riešenia zostaviť aj celé riešenie úlohy. Hlavným zdrojom informácií pri vysvetľovaní krokov návrhu DP algoritmu je opäť kniha *Introduction to Algorithms* [6]. Všetky kroky návrhu DP algoritmu budú vysvetlené na úlohe *Minimálny počet mincí* (úloha 1), ktorá je aj súčasťou výslednej aplikácie. Informácie k úlohe som čerpal z článku na stránke GeeksForGeeks [10].

Úloha 1. *Na vstupe máme mince rôznych hodnôt* ***C={ c1, c2, c3, ..., cN }****. Pre zjednodušenie si počet všetkých mincí označíme ako* ***N****. Predpokladajme, že mincí každej hodnoty je neobmedzený počet, teda môžeme použiť ľubovolný počet mincí* ***c1, c2, ...*** *Okrem mincí je na vstupe hodnota* ***V****, ktorá má vzniknúť súčtom hodnôt mincí. Pre súčet musí byť využitý čo najmenší počet mincí. Ktoré mince budú použité a aký je ich počet?*

2.3.1 Charakterizovanie štruktúry optimálneho riešenia úlohy

V prvom kroku je potrebné zadanú optimalizačnú úlohu analyzovať. Dôležité je určiť si jednoduché vstupné hodnoty, aby sa dalo k optimálnemu riešeniu dopracovať metódou „pokus-omyl“. Budeme teda testovať kombinácie všetkých možných hodnôt, na záver nám vzíde jedno alebo viacero optimálnych riešení s rovnakou hodnotou. Pri tomto naivnom zisťovaní výsledku budeme sledovať, či má úloha vlastnosti smerujúce k návrhu DP algoritmu.

Určíme si vstupné hodnoty úlohy:

- Mince: 1, 2, 3, 5
- Hodnota: 8

Na prvý pohľad je jasné, že riešením je použiť 8 mincí s hodnotou 1, ale toto riešenie určite nie je optimálne. Skúsime teda použiť mincu s hodnotou 2. Aby sme v súčte dostali hodnotu 8, musíme pridať ďalšie 3 mince s hodnotou 2, prípadne použiť mincu s hodnotou 5, a pridať ešte mincu s hodnotou 1. Vypíšeme si teda niektoré riešenia úlohy:

1. riešenie: 2, 2, 2, 2 - 4 mince
2. riešenie: 2, 2, 3, 1 - 4 mince
3. riešenie: 3, 3, 2 - 3 mince
4. riešenie: 3, 5 - 2 mince *(optimálne riešenie)*

Ak vyberieme mincu s hodnotou 2, musíme ďalej riešiť podproblém, kedy potrebujeme získať súčet 6. Pridáme mincu s hodnotou 2. Ďalší podproblém bude získať súčet 4, rôzne možnosti s rovnakým výsledkom ukazujú riešenia 1 a 2. Zaujímavejšie sú riešenia 3 a 4. Po pridaní mince s hodnotou 3 riešime podproblém, v ktorom potrebujeme získať hodnotu 5. Vybraním mincí s hodnotami 2 a 3 tento podproblém nebol vyriešený optimálne. Optimálne je vybrať mincu s hodnotou 5.

Pri riešení úlohy teda vznikajú podproblémy, ktoré je možné riešiť nezávisle, a ktorých optimálnym riešením získame aj optimálne riešenie pôvodného problému. Tým sme dokázali, že úloha má **optimálnu subštruktúru**.

Jednoducho by sa dal navrhnúť naivný algoritmus, ktorý bude postupne skúšať všetky možnosti. Ako je vidieť z vypísaných riešení, výpočet podproblému pre získanie hodnôt 6, 5, 4 aj 2 by sa opakoval. Po vypísaní všetkých riešení by bolo vidieť ďalšie duplicity. Tým sa potvrdila aj druhá vlastnosť a to **prekrývanie podproblémov**.

Pri tomto konkrétnom zadaní môže byť optimálne riešenie jasné na prvý pohľad. Je ním posledné riešenie, a teda súčet mincí s hodnotami 3 a 5. Hodnota optimálneho riešenia (počet mincí) je 2. Pri zadaní $\mathbf{C = \{\ 2,\ 4,\ 7,\ 13\ \}}$ a $\mathbf{V = 47}$ to také jednoduché nie je. Logicky skúsime vybrať najprv 3 mince s hodnotou 13 a jednu mincu s hodnotou 7, čím dostaneme hodnotu 46. Táto kombinácia k výsledku nevedie, pokračovali by sme teda ďalšou. V ďalšom kroku vytvoríme algoritmus, ktorý vykoná túto prácu za nás.

2.3.2 Definovanie hodnoty optimálneho riešenia pomocou rekurzie

Po charakterizovaní štruktúry optimálneho riešenia vzhľadom na podproblémy nasleduje 2. krok. Ten spočíva v návrhu rekurzívneho algoritmu. Využijeme teda prístup *zhora-dole*. Princípom je postupné rozkladanie riešenia na jednoduchšie operácie až k elementárnym krokom [31].

V prvom rade vezmeme do úvahy špecifický prípad, od ktorého sa odrazíme. Ak je zadaná hodnota rovná 0, aj počet mincí bude 0. Ak je hodnota väčšia ako 0, zoberieme postupne všetky mince, ktorých hodnota je menšia, alebo rovná požadovanej hodnote. Od požadovanej hodnoty sa odpočíta hodnota mince a nastane rekurzívne volanie pre novú požadovanú hodnotu, pričom sa inkrementuje počet použitých mincí.

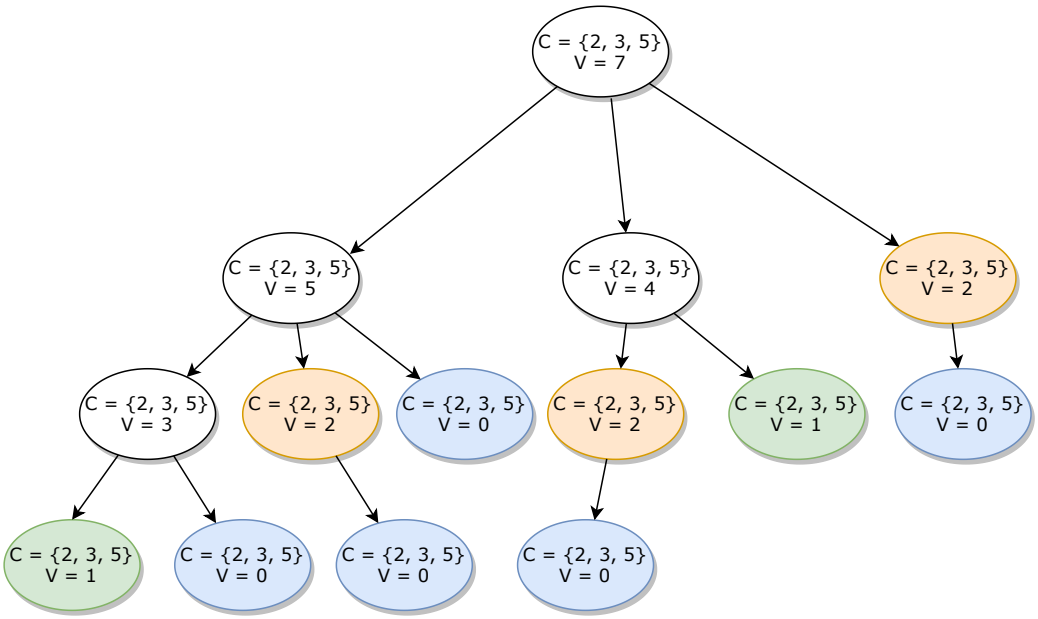
Algoritmus 1: minceRek(C, N, V) – rekurzívne riešenie úlohy *Minimálny počet mincí*

Vstup: C[], N, V (C - mince, N - počet mincí, V - hodnota)

```
1  if V = 0 then
2    return 0
3  end
4  vysledok = INT_MAX
5  for i = 0..N − 1 do
6    if C[i] ≤ V then
7      podvysledok = minceRek(C, N, V − C[i]) // Rekurzívne volanie
8      if podvysledok ≠ INT_MAX && podvysledok + 1 < vysledok then
9        vysledok = podvysledok + 1
10     end
11   end
12 end
13 return vysledok
```

Rekurzívne riešenie síce dáva správny výsledok, ale je veľmi neefektívne. V literatúre je uvedené, že časová zložitosť je exponenciálna. Vďaka experimentom v poslednej kapitole tejto práce 5.3 bola nájdená konkrétna funkcia zložitosti, a to $O(2^V)$.

V strome rekurzívnych volaní na obrázku 2.2 je vidieť, že metóda je volaná opakovane s rovnakými parametrami – **prekrývanie podproblémov**. Skúsime teda vytvoriť efektívnejší algoritmus, ktorý si bude pamätať výsledky podproblémov.



Obr. 2.2: Strom rekurzívnych volaní pri riešení úlohy *Minimálny počet mincí*. Vstupné parametre sú $\mathbf{C = \{\ 2, 3, 5\ \}}$ a $\mathbf{V = 7}$.

2.3.3 Výpočet hodnoty optimálneho riešenia prístupom zdola-hore

V predchádzajúcich krokoch bolo potvrdené, že úloha má potrebné vlastnosti, aby mohol nasledovať 3. krok – návrh algoritmu pomocou techniky dynamického programovania. Pri návrhu budeme vychádzať z rekurzívneho algoritmu. Je potrebné zaistiť, aby rovnaké podproblémy neboli vypočítavané opakovane, ich riešenia teda treba ukladať. Tým narastie priestorová zložitosť oproti rekurzívnemu riešeniu. Ide o tzv. *time-memory trade-off*. Je to kompromis, pri ktorom zrýchlime výpočet na úkor väčšieho využitia pamäte. Tento termín bol po prvý raz použitý v spojení s kryptoanalýzou pri zrýchlovaní výpočtu a ide práve o využitie dát uložených v pamäti za účelom zrýchlenia výpočtu [3]. Oproti pôvodnej exponenciálnej triede časovej zložitosti sa pri DP algoritme často dostaneme na polynomickú.

Priestorová zložitosť rekurzívneho algoritmu je v literatúre uvádzaná vždy ako veľkosť dátových štruktúr potrebných na uloženie vstupných hodnôt. Každý dobrý programátor by mal ale myslieť aj na réžiu potrebnú pre každé rekurzívne volanie, ktorá si vyžiada ďalší pamäťový priestor.

Pri DP algoritme je potrebné k veľkosti dátových štruktúr so vstupnými hodnotami pripočítať veľkosť štruktúry potrebnej na ukladanie výsledkov podproblémov. Ak je potrebné zostaviť celé optimálne riešenie úlohy, často treba pridať ďalšiu štruktúru na ukladanie prídavných informácií. Pri rekurzívnom algoritme získame iba hodnotu optimálneho rie-

šenia, celé riešenie nezostavujeme. Veľkosť prídavnej štruktúry preto nebude pri uvádzaní priestorovej zložitosti braná do úvahy.

Existujú 2 rôzne prístupy návrhu, pričom pri každom bude pracovať DP algoritmus odlišne:

- Prístup **zhora-dole** s využitím tzv. *„memoization“*[5]
- Prístup **zdola-hore** nazývaný aj **tabuľková metóda**

Prvý prístup spočíva v jednoduchej modifikácii rekurzívneho riešenia. Do algoritmu vytvoreného v predchádzajúcom kroku stačí pridať pole na ukladanie výsledkov podproblémov. Na mieste, kde by pôvodne nastalo rekurzívne volanie pre výpočet podproblému, najprv prebehne kontrola, či výsledok podproblému už nie je uložený. Ak áno, namiesto rekurzívneho volania sa použije uložený výsledok. Tým sa ušetrí čas potrebný pre opakovaný výpočet podproblému a všetkých jeho ďalších podproblémov.

Druhý prístup je síce tiež inšpirovaný rekurzívnym riešením, ale v podstate sa vytvára „nový“ algoritmus. Pri návrhu algoritmu **zdola-hore** sa snažíme od elementárnych krokov postupne dostať ku konečnému výsledku [31]. Teda presne naopak ako pri prvom prístupe. Podľa veľkosti postupujeme od najmenších podproblémov k väčším a postupne skladáme výsledok. Výsledky menších podproblémov sú pri riešení väčšieho podproblému vždy vypočítané a uložené, stačí ich použiť. Rovnako ako v prvom prístupe je teda výsledok každého podproblému počítaný iba raz.

Oba prístupy zabraňujú opakovanému výpočtu rovnakých podproblémov. Až na špecifické prípady, kedy nenastáva rekurzívne volanie pri riešení všetkých podproblémov, je časová zložitosť rovnaká. Nevýhodou prístupu **zhora-dole** môže byť réžia potrebná pre rekurzívne volanie metódy. Môže dôjsť napríklad až k pretečeniu zásobníku volaní [29]. Táto správa a aj výsledná aplikácia sa bude venovať prístupu **zdola-hore**.

Pri úlohe s minimálnym počtom mincí bude na ukladanie výsledkov podproblémov potrebné pole veľkosti zadanej hodnoty + 1. Na začiatku sú všetky položky v poli nastavené na hodnotu `INT_MAX`[6]. Výnimkou je číslo 0 na 0.-tom indexe (predpokladajme indexovanie od 0). Vychádzame z toho, že ak je zadaná hodnota rovná 0, aj riešenie je 0. Čísla na ďalších indexoch sa budú na konci výpočtu rovnať počtu mincí potrebných na vytvorenie hodnoty, ktorá sa rovná danému indexu. Hodnota optimálneho riešenia teda bude na poslednom indexe (požadovaná hodnota 5 znamená, že výsledok na piatom indexe v poli). DP algoritmus výpočtu (algoritmus 2) je celkom jednoduchý, pričom tento algoritmus je pripravený na zostavenie celého riešenia úlohy, ktoré bude vysvetlené v poslednom kroku.

[5] *Memoization* je výraz používaný v angličtine v spojení s technikou využívanou pri dynamickom programovaní. Technika spočíva v ukladaní hodnôt, ktoré budú využité neskôr v priebehu výpočtu.

[6] `INT_MAX` je konštanta, ktorá je často súčasťou programovacieho jazyka. Jej hodnota sa rovná najväčšiemu celému číslu, ktoré môže nadobudnúť celočíselná (`INT`) premenná. Túto konštantu budeme používať miesto hodnoty nekonečna kvôli jednoduchšiemu prechodu od pseudoalgoritmu k programovaciemu jazyku.

Algoritmus 2: minceDP(C, N, V) – DP riešenie úlohy *Minimálny počet mincí*

Vstup: C[], N, V (C - mince, N - počet mincí, V - hodnota)

```
1  pole[V + 1]
2  pomocnePole[V + 1] // Pomocné pole na zostavenie celého riešenia
3  pole[0] = 0, pole[1..N] = INT_MAX
4  for i = 1..V − 1 do
5    for j = 0..N do
6      if C[j] ≤ i then
7        podvysledok = C[i − C[j]]
8        if podvysledok ≠ INT_MAX && podvysledok + 1 < pole[i] then
9          pole[i] = podvysledok + 1
10         pomocnePole[i] = j
11       end
12     end
13   end
14 end
15 return pole[V]
```

Podproblémy sú riešené od najmenšieho po najväčší. Vonkajší cyklus zaistí, že bude najskôr vypočítaný optimálny výsledok pre hodnotu 1, potom 2 a tak ďalej. Vo vnútornom cykle sa vyhodnocuje, ktoré mince použiť, pričom do úvahy sa berú už vypočítané hodnoty poľa, ktoré predstavujú výsledky menších podproblémov. Nová hodnota bunky je vždy inkrementovaná hodnota niektorej z predchádzajúcich buniek. Vzdialenosť medzi počítanou bunkou, a bunkou z ktorej berie hodnotu závisí od toho, aké máme mince. Ak máme dostať hodnotu 4 a máme aj mincu s hodnotou 4, maximálna vzdialenosť, na ktorú sa vypočítava nová hodnota je 4.

Priestorová zložitosť sa oproti rekurzívnemu algoritmu zvýšila iba o veľkosť poľa potrebného na ukladanie výsledkov podproblémov. Oveľa výraznejší rozdiel je v časovej zložitosti. Keďže DP algoritmus obsahuje vnorený cyklus, jeho časová zložitosť je polynomická, konkrétne $O(N * V)$. Pre 10 mincí a požadovanú hodnotu 100 by maximálny počet opakovaní vnoreného cyklu bol 1000. Výpočet je neporovnateľne efektívnejší oproti rekurzívnemu riešeniu, kde by teoretický počet rekurzívnych volaní bol 2 ^ 100. Využitá pamäť je pri zložitých vstupných hodnotách v porovnaní so zrýchlením výpočtu zanedbateľná.

2.3.4 Zostavenie celého optimálneho riešenia úlohy

V niektorých prípadoch sa dá celé optimálne riešenie úlohy zostaviť iba využitím vypočítaných hodnôt optimálnych riešení podproblémov. Príkladom je úloha *Najdlhší spoločný podreťazec* (úloha 3), ktorá sa bude nachádzať aj vo výslednej aplikácii. Pri úlohe *Minimálny počet mincí* je potrebné vytvoriť ďalšie pole veľkosti zadanej hodnoty + 1. Následne, ak je do poľa s výsledkami podproblémov priraďovaná nová hodnota, do pomocného poľa je na rovnaký index priradená aktuálna hodnota počítadla **j** vnútorného cyklu. Je to hodnota indexu v zadanom poli mincí. Uloží sa teda index mince, ktorá bola použitá. Pomocou týchto indexov sa dá jednoducho zistiť, ktoré mince boli použité, a zostaviť tak celé optimálne riešenie. Algoritmus 3 využíva pole indexov použitých mincí z algoritmu 2 k zostaveniu celého optimálneho riešenia. Ide teda o doplnok k algoritmu 2.

Algoritmus 3: Zostavenie celého optimálneho riešenia úlohy *Minimálny počet mincí* (doplnok k algoritmu 2)

```
1  // Pole použitých minci o veľkosti počtu použitých mincí
2  pouziteMince[pole[V]]
3  // Na začiatku nastavíme zostatok na pôvodnú hodnotu V
4  zostatok = V
5  // Ak zostatok ≠ 0, pridávame mince
6  while zostatok ≠ 0 do
7     // Index použitej mince, pomocnePole je naplnené v algoritme 2
8     i = pomocnePole[zostatok]
9     // Vložíme mincu na danom indexe do poľa
10    pouziteMince.push(C[i])
11    // Od zostatku odpočítame hodnotu mince
12    zostatok = zostatok − C[i]
13 end
14 return pouziteMince
```

2.4 Príklady optimalizačných úloh riešiteľných dynamickým programovaním

V tejto sekcii sú uvedené a vysvetlené ostatné optimalizačné úlohy, ktoré sú súčasťou aplikácie. Podrobný rozbor všetkých úloh (ako tomu bolo v predchádzajúcej sekcii) by bol nad rámec tejto práce. Sústredíme sa preto hlavne na porovnanie rekurzívneho a DP algoritmu. Zdrojový kód alebo pseudokód, na základe ktorého je uvedená časová zložitosť algoritmu, sa nachádza v literatúre uvedenej pri popise príkladu.

2.4.1 Rezanie tyče

Optimalizačná úloha *Rezanie tyče* sa princípom rekurzívneho aj DP algoritmu podobá úlohe *Minimálny počet mincí*. Môžeme tu však nájsť pár odlišností, ktoré sú dôvodom, prečo je aj táto úloha súčasťou aplikácie. Informácie k úlohe som čerpal z knihy *Introduction to Algorithms* [6].

Úloha 2. *Majme oceľovú tyč dĺžky* ***L****. Túto tyč je potrebné narezať na menšie kusy. Na vstupe sú predajné ceny tyčí všetkých dĺžok od 1 po L:* ***P={p1, p2, p3, ..., pL}****. Na aké dĺžky treba narezať tyč, aby súčet cien tyčí, ktoré vzniknú, bol čo najvyšší?*

Ak rozrežeme tyč na dve časti, vzniknú 2 podproblémy – ako optimálne rozrezať prvý kus, a ako druhý kus. Ďalší rozrezaním vzniknú ďalšie podproblémy. Je teda potrebné postupne vypočítať optimálne riešenie pre všetky dĺžky tyče. Z nich nakoniec vzíde optimálne riešenie pre celú tyč. Konkrétne zadanie by mohlo vyzerať takto:

- Dĺžka tyče: 5
- Ceny: 1, 5, 6, 6, 9

Optimálne riešenia pre jednotlivé dĺžky tyče sú uvedené v tabuľke 2.1.

Časová zložitosť rekurzívneho algoritmu exponenciálne narastá s dĺžkou tyče. Funkcia zložitosti je $O(2^L)$.

Dĺžka tyče	1	2	3	4	5
Narezaná na dĺžky	1	2	1 + 2	2 + 2	2 + 3
Cena	1	5	6	10	11

Tabuľka 2.1: Riešenie úlohy *Rezanie tyče* pre ceny **P = { 1, 5, 6, 6, 9 }**.

V DP algoritme sa budú do jednorozmerného poľa ukladať najvyššie možné ceny pre všetky dĺžky tyče. Pri výbere hodnoty sa berú do úvahy ceny všetkých dĺžok vypočítaných predtým (to znamená menších od aktuálnej). Ku každej bunke sa pripočítajú ceny takých dĺžok tak, aby výsledná dĺžka neprekročila aktuálne vypočítavanú. Do bunky sa vyberie súčet tých dĺžok (ich cien), ktoré dávajú najvyššiu predajnú cenu. Výpočet hodnoty bunky je popísaný formulou 2.1.

$$T[l] = \begin{cases} 0 & ; l = 0 \\ \max\{P[i] + T[l-i-1]\};\ i:\ i < l & ; l > 0 \end{cases} \tag{2.1}$$

Celé riešenie sa dá získať veľmi podobne ako pri úlohe *Minimálny počet mincí*. Keďže algoritmus obsahuje vnorený cyklus, jeho časová zložitosť je vyjadrená funkciou $O(L^2)$.

Úlohy s podobným princípom vypĺňania tabuľky:

- Minimálny počet mincí (úloha 1, nachádza sa aj vo výslednej aplikácii)
- Problém batohu[7]
- Najdlhšia stúpajúca podsekvencia[8]

2.4.2 Najdlhší spoločný podreťazec

Pri úlohe *Najdlhší spoločný podreťazec* sú v DP algoritme výsledky podproblémov ukladané do dvojrozmerného poľa. Zaujímavosťou je, že pomocou hodnôt z rovnakého poľa sa dá zostaviť aj celé optimálne riešenie úlohy. Informácie k úlohe som čerpal z článku na stránke GeeksForGeeks [11].

Úloha 3. *Na vstupe sú dva reťazce.* ***X*** *s dĺžkou* ***L1****, a* ***Y*** *s dĺžkou* ***L2****. Aký je ich najdlhší spoločný podreťazec?*

Jednoduchým riešením tejto úlohy by bolo zobrať každý podreťazec z prvého reťazca a zistiť, či sa tento podreťazec nachádza aj v druhom reťazci. V prvom reťazci bude $L1^2$ podreťazcov. Pri využití *Knuth-Morris-Prattovho*[9] algoritmu na vyhľadávanie všetkých podreťazcov v druhom reťazci bude časová zložitosť tohoto riešenia $O(L1^2 * L2)$.

Pri rekurzívnom algoritme, ak by dĺžka jedného alebo druhého reťazca bola rovná 0, aj výsledok by bol 0, pretože v jednom z reťazcov nebude žiadny podreťazec. Ak algoritmus narazí na zhodu znakov, pripočíta k výsledku hodnotu 1 a pokračuje porovnaním nasledujúcich dvoch znakov. Ak zhoda nenastane, nastáva rekurzívne volanie pre porovnanie ďalšieho znaku v prvom reťazci a ďalšieho znaku v druhom reťazci. Z týchto hodnôt sa

[7] https://www.gatevidyalay.com/0-1-knapsack-problem-using-dynamic-programming-approach/

[8] https://www.geeksforgeeks.org/longest-increasing-subsequence-dp-3/

[9] *Knuth–Morris–Prattov* alebo **KTM** je efektívny algoritmus na vyhľadávanie v texte. Ak máme reťazec dĺžky **L1** a text dĺžky **L2**, pričom $L1 \leq L2$, tento algoritmus dokáže potvrdiť alebo vyvrátiť výskyt reťazca v texte v čase $O(L2)$ [6].

vyberie maximum. Algoritmus je síce jednoduchý, ale veľmi neefektívny. V literatúre sa neuvádza jeho časová zložitosť. V tomto prípade sa dá oprieť o myšlienku, ktorá bude použitá aj pri úlohe *Editačná vzdialenosť* 2.4.3. V algoritme nastáva rekurzívne volanie na 3 miestach, základ bude teda číslo **3**. Minimálne jeden z reťazcov je pri rekurzívnom volaní vždy skrátený o 1 znak, časovú zložitosť teda vyjadruje funkcia $O(3^{L1+L2-1})$.

Princíp DP algoritmu spočíva v hľadaní najdlhšieho spoločného sufixu (znaky na konci reťazca) v podreťazcoch oboch reťazcov. Dĺžky sufixov sa budú ukladať do dvojrozmerného poľa (tabuľky). Riadky tabuľky symbolizujú znaky reťazca **X**. Stĺpce tabuľky reprezentujú znaky reťazca **Y**. Ak sa znak v stĺpci a riadku nerovná, do bunky sa vloží hodnota 0. Ak sa rovnajú, tak sa do bunky vloží hodnota 1, ak ide o prvý znak jedného alebo druhého reťazca. Ak nejde o prvý znak, inkrementuje sa predchádzajúca hodnota na diagonále (inkrementuje sa sufix). V tomto je zmena oproti riešeniu uvedenému v literatúre, kde má tabuľka jeden riadok a stĺpec navyše s nulovými hodnotami, aby nebolo treba extra riešiť zhodu pri prvom znaku reťazca. Výpočet hodnoty bunky je popísaný formulou 2.2.

$$T[i][j] = \begin{cases} 0 & ; X[i] \neq Y[j] \\ 1 & ; X[i] = Y[j] \wedge (i = 0 \vee j = 0) \\ T[i-1][j-1] + 1 & ; X[i] = Y[j] \wedge i > 0 \wedge j > 0 \end{cases} \tag{2.2}$$

Pri DP riešení tejto úlohy nie je potrebná žiadna štruktúra navyše k zostaveniu celého optimálneho riešenia. Stačí zistiť, v ktorej bunke tabuľky sa nachádza maximálna hodnota. Keďže ide o dĺžku najdlhšieho spoločného sufixu, na tomto indexe sa nachádza posledný znak sufixu. Od indexu stačí odpočítať hodnotu bunky a dostaneme počiatočný index sufixu. Stačí sa už iba pozrieť na zadaný reťazec (buď jeden alebo druhý, záleží od toho z ktorej dimenzie poľa bol vybraný index) a vybrať reťazec od začiatočného po konečný index. Príklad vyplnenej tabuľky je na obrázku 2.3.

	n	a	t	e	r	a	z
i	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0
a	0	1	0	0	0	1	0
r	0	0	0	0	1	0	0
a	0	1	0	0	0	2	0
z	0	0	0	0	0	0	3

Obr. 2.3: Riešenie úlohy *Najdlhší spoločný podreťazec* pre reťazce **X = 'ibaraz'** a **Y = 'nateraz'**. Najdlhší spoločný podreťazec je **'raz'** a má dĺžku **3**. Tabuľka pochádza z výslednej aplikácie.

Keďže ide o jednoduché vypĺňanie dvojrozmerného poľa, časová zložitosť DP algoritmu je $O(L1 * L2)$. Pre túto úlohu existuje dokonca ešte efektívnejšie riešenie ako DP algoritmus. Pri využití tzv. *sufixového stromu* sa dá úloha vyriešiť v čase $O(L1 + L2)$ [13]. Keďže sa táto práca zaoberá dynamickým programovaním, toto riešenie nebude bližšie preberané.

Úlohy s podobným princípom vypĺňania tabuľky:

- Najdlhší palindrom v reťazci[10]

2.4.3 Editačná vzdialenosť

Rovnako ako pri úlohe *Najdlhší spoločný podreťazec* sú aj v DP algoritme úlohy *Editačná vzdialenosť* hodnoty ukladané do dvojrozmerného poľa. Líši sa však veľkosť tohoto poľa a princíp výpočtu novej hodnoty. Tiež zostavenie celého optimálneho riešenia úlohy je v tomto prípade podstatne zložitejšie. Informácie k úlohe som čerpal z článku na stránke Tutorial Horizon [32].

Úloha 4. *Na vstupe sú dva reťazce.* ***X*** *s dĺžkou* ***L1*** *a* ***Y*** *s dĺžkou* ***L2****. K dispozícii sú 3 operácie:* ***pridanie****,* ***odobratie*** *a* ***nahradenie*** *znaku. Pomocou týchto operácií je potrebné upraviť prvý reťazec* ***X*** *tak, aby sa rovnal druhému reťazcu* ***Y****. Počet operácií musí byť čo najnižší a všetky operácie majú rovnakú váhu. Ktoré operácie je treba použiť?*

Rekurzívne riešenie bude spočívať v porovnávaní oboch reťazcov znak po znaku. Prakticky je jedno, či zľava alebo sprava, častejšie je však uvádzaný priechod sprava. Začneme teda posledným znakom. Ak sa posledný znak v prvom reťazci rovná poslednému znaku v druhom reťazci, znak ignorujeme a pokračujeme rekurzívne pre dĺžky L1 - 1 a L2 - 2. Druhý prípad je, že sa znaky nerovnajú (prípadne, že v druhom reťazci už nie sú ďalšie znaky). V tomto prípade je treba vyskúšať všetky 3 operácie:

- **Vložíme** na koniec prvého reťazca rovnaký znak ako je na konci druhého reťazca. Dĺžka prvého reťazca je teraz L1 + 1, rekurzívne pokračujeme pre dĺžky reťazcov L1 a L2 - 1.
- **Odoberieme** posledný znak z prvého reťazca. Dĺžka prvého reťazca je L1 - 1, rekurzívne pokračujeme pre dĺžky L1 - 1 a L2.
- **Nahradíme** posledný znak, aby sa posledné znaky v oboch reťazcoch rovnali a rekurzívne pokračujeme pre dĺžky L1 - 1 a L2 - 1.

Z týchto troch vyberieme tú možnosť, ktorá viedla k optimálnemu riešeniu (bolo použitých najmenej operácií).

Časová zložitosť rekurzívneho riešenia bude opäť exponenciálna. Keďže na výber sú 3 operácie, základ bude číslo **3**. V literatúre nie je uvedený presný exponent, preto sa opriem o myšlienku z diskusného fóra **StackOverflow**. Keďže minimálne jeden z reťazcov je pri rekurzívnom volaní vždy skrátený o 1 znak, časovú zložitosť by mala vyjadrovať funkcia $O(3^{L1+L2-1})$ [30].

DP algoritmus porovnáva reťazce postupne z ľavej strany. Začne reťazcami s dĺžkou 0 a postupuje všetkými kombináciami dĺžok reťazcov. Veľkosť tabuľky je $(L1 + 1) * (L1 + 2)$, pretože prvý riadok a stĺpec symbolizujú prázdne reťazce. Ak je reťazec **X** prázdny, je treba vložiť všetky znaky druhého reťazca, hodnota bunky je teda poradové číslo stĺpca. Ak je

[10] https://www.geeksforgeeks.org/longest-palindrome-substring-set-1/

reťazec **Y** prázdny, je treba odstrániť všetky znaky prvého reťazca, hodnota bunky je teda poradové číslo riadku. Ak sa znaky rovnajú, do bunky sa vloží predchádzajúca hodnota z diagonály - nie je potrebná žiadna operácia. Ak sa znaky nerovnajú, inkrementuje sa minimálna hodnota z nasledujúcich buniek:

- predchádzajúci stĺpec – **pridanie** znaku
- predchádzajúci riadok – **odstránenie** znaku
- predchádzajúca hodnota z diagonály – **nahradenie** znaku

Výpočet hodnoty bunky je popísaný formulou 2.3.

$$T[i][j] = \begin{cases} j & ; i = 0 \\ i & ; j = 0 \\ T[i-1][j-1] & ; X[i-1] = Y[j-1] \\ 1 + \min\{T[i][j-1],\ T[i-1][j],\ T[i-1][j-1]\} & ; X[i-1] \neq Y[j-1] \end{cases} \tag{2.3}$$

Na konci výpočtu sa bude počet potrebných operácií nachádzať v bunke `tabuľka[L1][L2]`. K zisteniu konkrétnych operácií, ktoré boli použité je treba pridať spätný priechod tabuľkou. Inšpirovať sa dá riešením v jazyku Java, ktoré je verejne prístupné na stránke GitHub [14]. Príklad vyplnenej tabuľky je na obrázku 2.4.

	Prázdny	P	r	i	d	O	d	s
Prázdny	0	1	2	3	4	5	6	7
P	1	0	1	2	3	4	5	6
r	2	1	0	1	2	3	4	5
i	3	2	1	0 + 1	1	2	3	4
O	4	3	2	1	1	1	2	3
d	5	4	3	2	1	2	1	2
s	6	5	4	3	2	2	2	1 + 1
t	7	6	5	4	3	3	3	2

Obr. 2.4: Riešenie úlohy *Editačná vzdialenosť* pre reťazce **X = 'PriOdst'** a **Y = 'PridOds'**. Je potrebné **pridať** znak **'d'** a **odstrániť** znak **'t'**. Tabuľka pochádza z výslednej aplikácie.

Keďže ide opäť o vypĺňanie dvojrozmerného poľa, časová zložitosť DP algoritmu je polynomická. Vyjadruje ju funkcia $O((L1+1)*(L2+1))$.

Úlohy s podobným princípom vypĺňania tabuľky:

- Najdlhšia palindromická podsekvencia[11] (v pôvodnom reťazci sú ponechané len tie znaky, ktoré tvoria palindrom, pričom novovzniknutý reťazec sa nazýva palindromická podsekvencia)
- Najdlhšia spoločná podsekvencia[12]

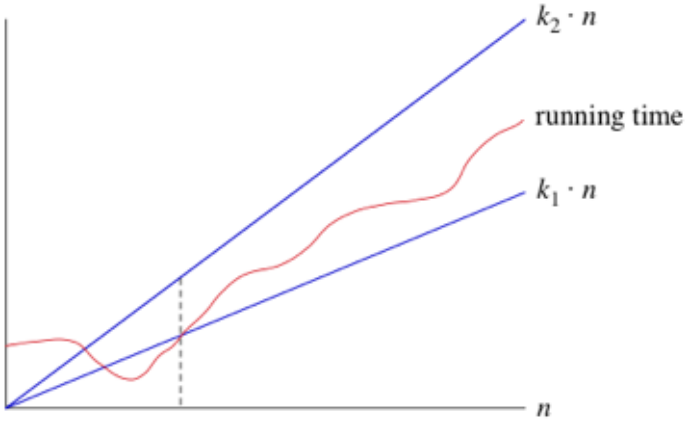
2.4.4 Optimálny binárny vyhľadávací strom

DP algoritmus, ktorý rieši úlohu *Optimálny binárny vyhľadávací strom*, sa od ostatných úloh v aplikácii líši v tom, že obsahuje až 3 vnorené cykly, pričom výsledky podproblémov sú ukladané do „iba“ dvojrozmerného poľa. K zostaveniu celé optimálneho riešenia je navyše potrebné ďalšie pole rovnakej veľkosti na ukladanie pomocných hodnôt. Informácie k úlohe som čerpal z článku na stránke GeeksForGeeks [12].

Úloha 5. *Na vstupe je pole vyhľadávacích kľúčov* ***K={ k1, k2, k3, …, kN }****, ktoré sú vzostupne zoradené. Pre zjednodušenie si počet všetkých kľúčov označíme ako* ***N****. Druhé pole na vstupe obsahuje počet vyhľadaní kľúčov z prvého poľa* ***F = { f1, f2, f3, …, fN }****. Index kľúča v prvom poli sa rovná indexu počtu vyhľadaní v druhom poli. Je potrebné vytvoriť binárny vyhľadávací strom, ktorý bude obsahovať všetky kľúče. Súčet cien všetkých vyhľadaní musí byť čo najnižší. Cena vyhľadania sa násobí s každou úrovňou stromu. 1. úroveň = počet vyhľadaní * 1, 2. úroveň = počet vyhľadaní * 2 atď.*

Rekurzívne riešenie bude spočívať v postupnom dosadzovaní všetkých kľúčov ako koreň stromu, koreň podstromu atď. Pre všetky takéto podstromy sa spočíta cena. Ak bude každý podstrom optimálny, získame optimálnu štruktúru celého stromu.

V tomto prípade bude výnimočne časová zložitosť vyjadrená pomocou *Theta* notácie. Podľa [20] by sa dalo jednoducho povedať, že táto notácia udáva presnú hodnotu zložitosti. V skutočnosti to ale nebude presná hodnota, definícia je trochu zložitejšia. Majme funkciu zložitosti $\theta(N)$. Ak bude N dosť veľké, existujú konštanty $k1$ a $k2$, pre ktoré časová zložitosť bude najmenej $k1 * N$ a najviac $k2 * N$ [19]. Priebeh funkcie je znázornený na obrázku 2.5.



Obr. 2.5: Priebeh funkcie zložitosti $\theta(N)$ (prevzaté z [19]).

[11] https://www.geeksforgeeks.org/longest-palindromic-subsequence-dp-12/
[12] https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/

Rekurzívny algoritmus testuje postupne všetky možné stromy. Časová zložitosť by sa teda mala blížiť počtu stromov, ktorý je vyjadrený v dokumente na stránkach Univerzity Carnegieho-Mellonových práve pomocou Theta notácie – $\theta(4^N * N^{-3/2})$ [35].

DP algoritmus najprv do buniek na diagonále tabuľky priradí počty vyhľadaní jednotlivých kľúčov. To znamená, že na diagonále sú ceny vyhľadávaní v stromoch s jedným uzlom (počet vyhľadaní uzlu je výsledná cena stromu s jediným uzlom). Následne budú do buniek nad diagonálou priraďované ceny optimálnych stromov pre všetky kombinácie uzlov. Najprv sa tvoria stromy s veľkosťou 2 uzly (riadok 1 – stĺpec 2, riadok 2 – stĺpec 3, ...), potom 3 uzly (riadok 1 – stĺpec 3, riadok 2 – stĺpec 4) atď. Bunky pod diagonálou zostanú prázdne. Pri výpočte hodnoty bunky sa dosadzujú postupne všetky uzly v aktuálnom strome ako korene, a počíta sa cena takéhoto stromu. Na konci výpočtu bude v bunke hodnota toho stromu, ktorý mal najnižšiu cenu vyhľadaní. Výpočet hodnoty bunky je popísaný formulou 2.4.

$$T[j][col] = \begin{cases} F[j] & ; j = col \\ \min\left\{T[j][r-1] + T[r+1][col] + \sum_{k=j}^{col} F[k]\right\} & ; X[i-1] \neq Y[j-1] \\ & ; i : i \geq 2 \wedge i \leq N \\ & ; j : j \leq N - i + 1 \\ & ; col : j + i - 1 \\ & ; r : r \geq j \wedge r \leq col \end{cases} \tag{2.4}$$

Výsledná cena optimálneho stromu sa na konci nachádza v poslednej bunke prvého riadku. Do tabuľky sa však ukladajú iba ceny. Ak by sme chceli zostaviť celý strom, museli by sme do ďalšej tabuľky ukladať aj použité kľúče. Kľúč predstavuje koreň podstromu, takže sme schopní na základe kľúčov z tabuľky zostaviť strom.

Ako koreň celého stromu zvolíme kľúč z bunky s výslednou cenou. Vyberieme ďalší kľúč z poľa vľavo od koreňového kľúča. Ak je takýto kľúč jeden, rovno ho umiestnime do ľavého podstromu a pokračujeme tvorením pravého podstromu. Dôvodom je, že kľúče v poli sú zoradené vzostupne, teda v ľavom podstrome už nebude žiadny iný kľúč. Ak je vľavo kľúčov viac, vyberieme dva a pozrieme sa do tabuľky, ktorý z týchto dvoch kľúčov bol vybraný ako koreň podstromu. Ak sa minú všetky kľúče naľavo, pokračujeme kľúčmi napravo, až pokým nebudú v strome umiestnené všetky kľúče. Na obrázku 2.6 je vyplnená tabuľka (obsahuje ceny aj kľúče) a strom zostavený podľa tabuľky. V DP algoritme sa nachádzajú 3 vnorené cykly, časová zložitosť je teda $O(N^3)$.

Úlohy s podobným princípom vypĺňania tabuľky:

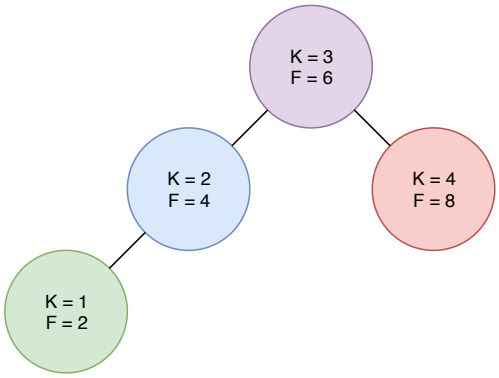
- Reťazové násobenie matíc[13]
- Predikcia sekundárnej štruktúry RNA[14] (predikcia je založená na hľadaní palindromatických štruktúr, zahŕňajúcich najväčšie možné množstvo znakov vstupného reťazca)

[13] http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Dynamic/chainMatrixMult.htm

[14] https://www.cs.cmu.edu/~02710/Lectures/RNALecture2015.pdf

	Kľúč 1 (2)	Kľúč 2 (4)	Kľúč 3 (6)	Kľúč 4 (8)
Kľúč 1 (2)	2 (0)	8 (1)	20 (1)	36 (2)
Kľúč 2 (4)		4 (1)	14 (2)	30 (2)
Kľúč 3 (6)			6 (2)	20 (3)
Kľúč 4 (8)				8 (3)

(a) Vyplnená tabuľka z výslednej aplikácie.



(b) Optimálny binárny vyhľadávací strom vytvorený podľa tabuľky.

Obr. 2.6: Riešenie úlohy *Optimálny binárny vyhľadávací strom* pre **K = { 1, 2, 3, 4 }** a **F = { 2, 4, 6, 8 }**. Hodnota v zátvorke je index v poli kľúčov - symbolizuje ktorý kľúč bol použitý ako koreň stromu pre daný podproblém. Pomocou týchto pomocných hodnôt sa dá vytvoriť celý strom.

Kapitola 3

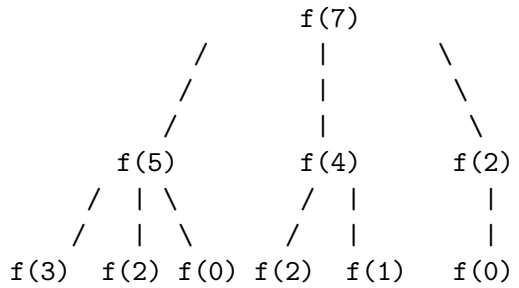
Návrh webovej aplikácie

Existuje množstvo webových stránok, ktoré sa zaoberajú dynamickým programovaním. Prvá časť tejto kapitoly je zameraná na rozbor týchto stránok. V ďalšej časti budú aj na základe tohto rozboru uvedené požiadavky na novú webovú aplikáciu, ktorá má byť výsledkom tejto práce. Nasleduje návrh užívateľského rozhrania a v záverečnej časti výber moderných technológií, na ktorých bude aplikácia postavená.

3.1 Webové stránky so zameraním na dynamické programovanie

Na internete v súčasnosti možno nájsť veľký počet webových stránok a aplikácií, ktoré sa zaoberajú dynamickým programovaním. Niektoré sa venujú rôznym technikám návrhu algoritmov, prípadne aj ďalším odvetviach z oblasti informačných technológií. Nájdu sa aj také, ktoré sa sústreďujú výhradne na dynamické programovanie. Jedny aj druhé ponúkajú užitočné riešenia, ktoré sa budú dať využiť.

GeeksforGeeks[1] je portál, ktorý ponúka okrem iného aj široké spektrum informácií z oblasti algoritmizácie a programovania. Z hľadiska DP ponúka okrem základných konceptov aj pokročilé postupy, ktorými možno ešte viac zefektívniť DP algoritmy. Môžeme tu nájsť vyše 300 optimalizačných úloh rôznej náročnosti. Pri každej je uvedený najprv naivný rekurzívny algoritmus, prípadne iné jednoduché, ale neefektívne riešenie úlohy. Uvedený býva aj strom rekurzívnych volaní. Často je však nakreslený len pomocou znakov. Strom nakreslený týmto spôsobom je na obrázku 3.1.



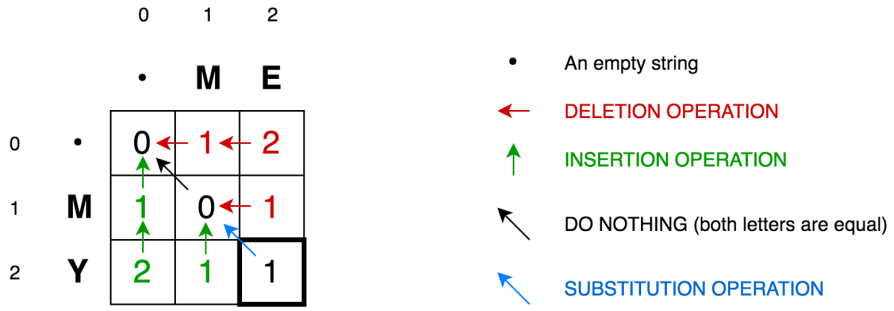
Obr. 3.1: Strom rekurzívnych volaní nakreslený pomocou znakov.

[1] https://www.geeksforgeeks.org/

Takéto riešenie z môjho pohľadu nie je úplne ideálne, strom nakreslený pomocou vektorového grafického editoru by bol lepšie čitateľný.

Po rekurzívnom nasleduje vysvetlenie DP algoritmu. K algoritmom sú dostupné zdrojové kódy v rôznych programovacích jazykoch. Dajú sa jednoducho kopírovať, prepisovať a dokonca aj spustiť. Často sú uvedené aj zložitosti algoritmov a vysvetlenie na videu. K uvedeným riešeniach môžu čitatelia portálu pridať ďalšie užitočné poznámky v diskusii. Podobných portálov, ktoré poskytujú teóriu k dynamickému programu je niekoľko. Spomenúť môžeme napríklad **TutorialsPoint**[2] alebo **Techie Delight**[3].

DZone[4] je jedna z najväčších online komunít softvérových vývojárov. Prebiehajú tu diskusie o najnovších trendoch, moderných technológiách pre vývoj ale aj o rôznych technikách návrhu algoritmov, ku ktorým patrí aj dynamické programovanie. Práve v jednom článku, ktorý sa venuje základom DP a jeho podobnosti s metódou Rozdeľuj a panuj, môžno nájsť dôležitý prvok, ktorý by bolo dobré použiť vo výslednej aplikácii. Ide o grafické znázornenie tabuľky, do ktorej sa ukladajú výsledky podproblémov. Tabuľka je na obrázku 3.2.

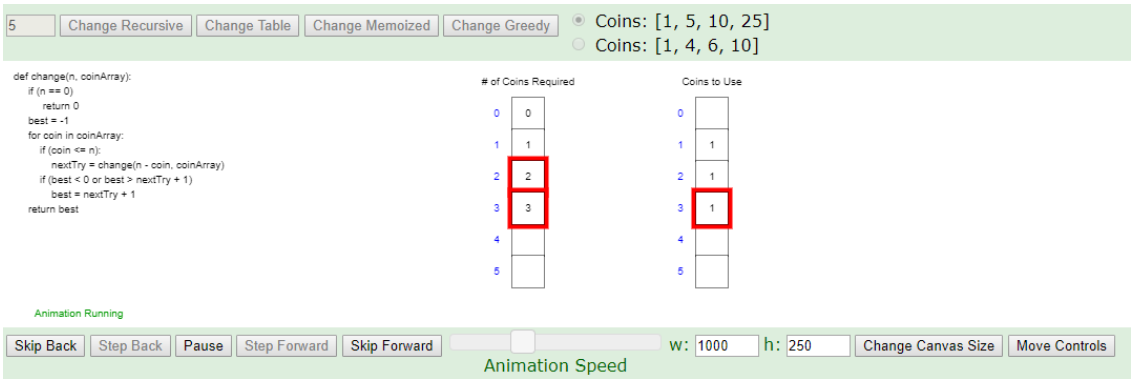


Obr. 3.2: Tabuľka znázorňuje dátovú štruktúru, do ktorej sú ukladané výsledky podproblémov pri optimalizačnej úlohe *Editačná vzdialenosť*. Šipky ukazujú, od ktorých buniek sa môže odvíjať hodnota novej bunky. Obrázok navyše obsahuje aj popis jednotlivých operácií priradených k šipkám (prevzaté z [22]).

Na tejto stránke navyše možno nájsť odkaz na aplikáciu **DP Visualizer**[5], ktorá dokáže vizualizovať riešenie optimalizačnej úlohy pomocou DP algoritmu. Aplikácia je veľmi užitočná, ide však o prístup **zhora-dole**, preto sa jej nebudeme bližšie venovať.

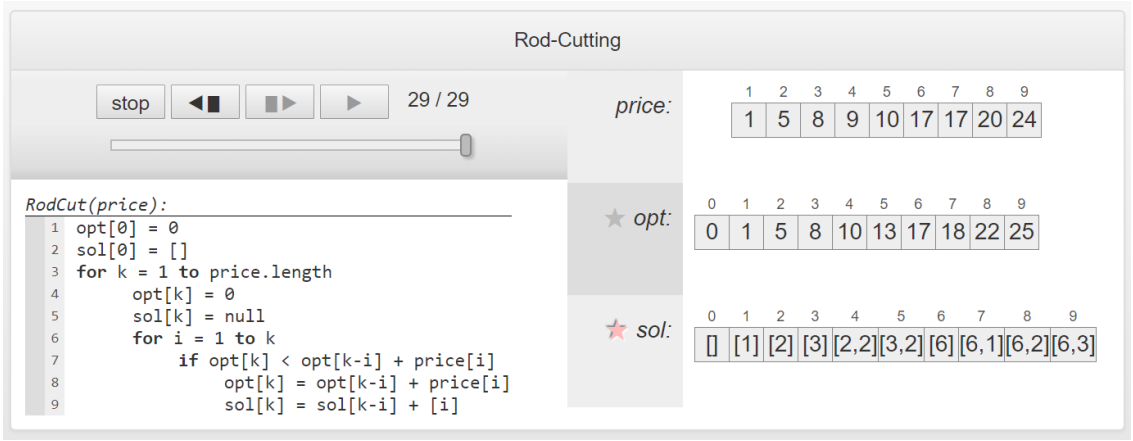
Na webových stránkach Davida Gallesa zo sanfranciskej univerzity sú dostupné aplikácie pre vizualizáciu rôznych typov algoritmov – **Data Structure Visualizations**[6]. Pri DP algoritme je vždy uvedený jeho pseudokód a po zadaní vstupných hodnôt a zvolení DP prístupu **zdola-hore** sa spustí vypĺňanie tabuľky (obrázok 3.3). Dá sa zvoliť aj animácia pre prístup **zhora-dole**, prípadne aj tzv. *„Hladný“ algoritmus*[7].

[2] https://www.tutorialspoint.com/introduction-to-dynamic-programming
[3] https://www.techiedelight.com/rot-cutting/
[4] https://dzone.com/
[5] http://easyhard.github.io/dpv/
[6] https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
[7] https://xlinux.nist.gov/dads//HTML/greedyalgo.html



Obr. 3.3: Priebeh riešenia úlohy *Vydávanie mincí* tabuľkovou metódou **zdola-hore**. Okrem dynamickej tabuľky je dostupný aj popis algoritmu. Rýchlosť animácie sa dá prispôsobiť, rovnako aj veľkosť plátna (prevzaté z [9]).

Vamonos je knižnica na vizualizáciu algoritmov a dátových štruktúr v internetových prehliadačoch. Na jej stránkach možno nájsť aj demá pre 3 úlohy riešené dynamickým programovaním[8], ktoré využívajú túto knižnicu. Užívateľské rozhranie je rozdelené na 2 časti, a to zdrojový kód a tabuľky. Princíp spočíva v krokovaní zdrojového kódu a súčasného vypĺňania tabuliek. Ako možno vidieť na obrázku 3.4, jedna tabuľka obsahuje hodnoty optimálnych riešení podproblémov a druhá aj celé riešenia.



Obr. 3.4: Priebeh riešenia úlohy *Rezanie tyče*. Pre vizualizáciu je využitá knižnica **Vamonos**. V ľavej časti sú ovládacie prvky a zdrojový kód, napravo tabuľky so vstupnými hodnotami, výsledkami optimálnych riešení podproblémov a aj tabuľka s celými riešeniami (prevzaté z [33]).

Všetky uvedené riešenia sú voľne dostupné pre všetkých a dajú sa pomerne rýchlo nájsť. Portálov, ktoré sa sústreďujú na teóriu je viac, praktických demo aplikácií menej. K vytvoreniu požiadaviek na finálnu aplikáciu, ktoré sú popísané v nasledujúcej sekcii, veľmi pomohol práve tento prieskum existujúcich riešení.

[8] http://rosulek.github.io/vamonos/demos/

3.2 Požiadavky kladené na novú webovú aplikáciu

Hlavnou požiadavkou na výslednú aplikáciu je grafické znázornenie priebehu DP algoritmu **zdole-hore**. V tomto prípade sa dá inšpirovať demo aplikáciami na stránkach knižnice **Vamonos** alebo aplikáciou **Data Structure Visualizations** od Davida Gallesa. Základom teda bude tabuľka s výsledkami podproblémov. Bude vypĺňaná dynamicky, aby bolo jasné, ako algoritmus funguje. Aby užívateľ vedel, prečo sa do buniek ukladajú práve také a také hodnoty, bude tu uvedená formula, ktorá popisuje výpočet hodnôt. Samozrejmosťou je vyznačovanie buniek rôznymi farbami. Bude na výber viacero rýchlostí a tiež vypĺňanie tabuľky „krok po kroku“. Na konci výpočtu sa zobrazí hodnota optimálneho riešenia aj celé optimálne riešenie úlohy.

Animovaná tabuľka bez vysvetlenia nemá veľkú výpovednú hodnotu. Pre užívateľa by nebolo príjemné, ak by si musel hľadať všetky informácie o tom, ako algoritmus funguje, sám. Navyše algoritmov, ktoré riešia danú úlohu môže byť viacero. Práve ten, ktorý by užívateľ našiel nemusí úplne sedieť s algoritmom, s ktorým pracuje aplikácia. Nevyhnutný je preto úvod do dynamického programovania a teoretická časť ku každej úlohe. Musí byť jasné, aké vstupy je možné zadať u danej úlohy, aké hodnoty sa vyberajú do aktuálnej bunky, v ktorej bunke bude na konci hodnota optimálneho riešenia atď. Tu sa dá inšpirovať portálmi s teóriou, napríklad **GeeksforGeeks**. Pri rekurzívnom algoritme bude uvedený strom rekurzívnych volaní nakreslený vo vektorovom grafickom editore. Pri DP algoritme bude zasa tabuľka s vysvetlením ako sa počítajú nové hodnoty buniek, podobne ako v článku na **Dzone**. Okrem toho budú uvedené zdrojové kódy v jazyku C a zložitosti algoritmov. Uvedené budú aj príklady optimalizačných úloh s podobným princípom vypĺňania tabuľky.

Úlohou aplikácie je aj poukázať na výhody, prípadne nevýhody dynamického programovania. Ku každej úlohe bude možné vygenerovať štatistiky. Tie budú zakreslené do grafov a tabuliek. Pôjde o porovnanie priestorovej a časovej zložitosti rekurzívneho a DP algoritmu. Budú uvedené maximálne teoretické, ale aj reálne hodnoty pre konkrétne vstupy. Okrem preddefinovaných bude možné zadať aj vlastné vstupné hodnoty. Uvedené bude aj zhodnotenie štatistík. Túto funkcionalitu neponúka žiadne zo súčasných riešení.

Aplikácia bude lokalizovaná okrem slovenčiny aj do angličtiny. Dá sa povedať, že angličtina je jazyk informatiky, aplikácia teda môže byť využitá širokou komunitou.

Zhrnutie požiadaviek na výslednú aplikáciu:

- Teoretický úvod do dynamického programovania
- Konkrétne príklady optimalizačných úloh
- Teória ku každej úlohe
- Strom rekurzívnych volaní
- Zdrojové kódy v jazyku C rekurzia/DP
- Zložitosti algoritmov rekurzia/DP
- Príklady optimalizačných úloh s podobným princípom vypĺňania tabuľky
- Grafické znázornenie priebehu DP algoritmu popísané aj formulou
- Štatistiky rekurzia/DP v grafoch a tabuľke + zhodnotenie štatistík
- Lokalizácia do slovenčiny a angličtiny

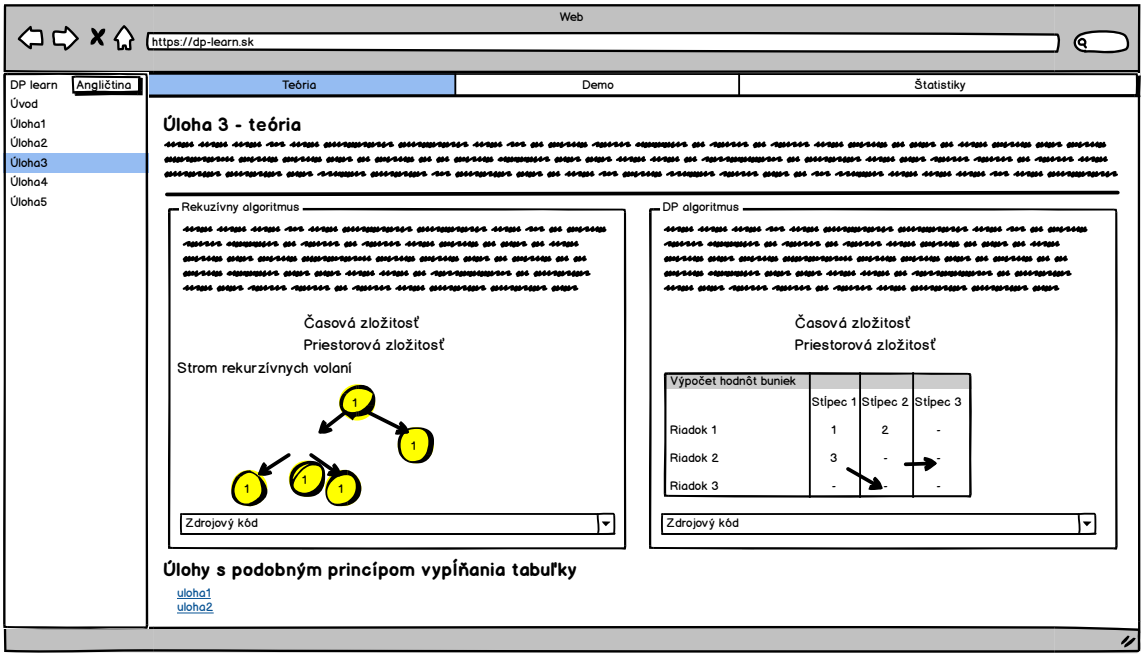
3.3 Návrh užívateľského rozhrania

Aby bola aplikácia prehľadná a medzi jednotlivými časťami sa dalo rýchlo a intuitívne prechádzať, je treba ju rozdeliť do niekoľkých sekcií. V ľavom hornom rohu obrazovky bude názov aplikácie a **prepínač jazykov**. Pod ním sa bude nachádzať **menu**, ktoré bude vždy viditeľné. Prvé tlačidlo v menu zobrazí informácie o aplikácii a jednoduchý úvod do dynamického programovania. Pomocou ostatných tlačidiel sa bude dať prepínať medzi jednotlivými optimalizačnými úlohami. Tento obsah sa zobrazí v hlavnom okne aplikácie.

Pri každej optimalizačnej úlohe sa bude dať v hlavnom okne prepínať medzi tromi záložkami:

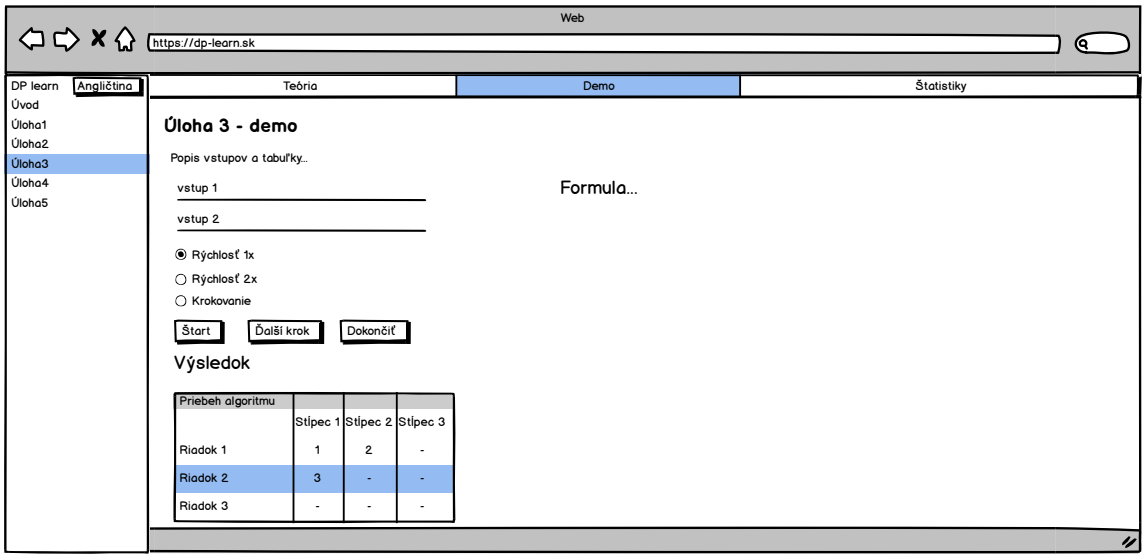
- Teória
- Demo
- Štatistiky

Teória je statická stránka, ktorá bude poskytovať dôležitú teóriu k danej úlohe. Vo vrchnej časti bude zadanie úlohy s konkrétnym príkladom. Nasleduje vedľa seba popis rekurzívneho a DP algoritmu, a tiež ich časová a priestorová zložitosť. Pri rekurzívnom algoritme bude uvedený strom rekurzívnych volaní. Pri DP algoritme bude na obrázkoch vysvetlené, akým spôsobom sú vyberané/vypočítavané nové hodnoty do buniek tabuľky. Tabuľka bude vyzerať rovnako ako v záložke **Demo**. Zdrojové kódy sa budú dať zobraziť a skryť. Príklady ďalších optimalizačných úloh budú zobrazené v zozname, pričom pôjde o klikateľné odkazy, ktoré užívateľa presmerujú na webovú lokalitu, kde je táto úloha preberaná. Mockup záložky je na obrázku 3.5.



Obr. 3.5: Mockup aplikácie pri otvorenej záložke `Teória`.

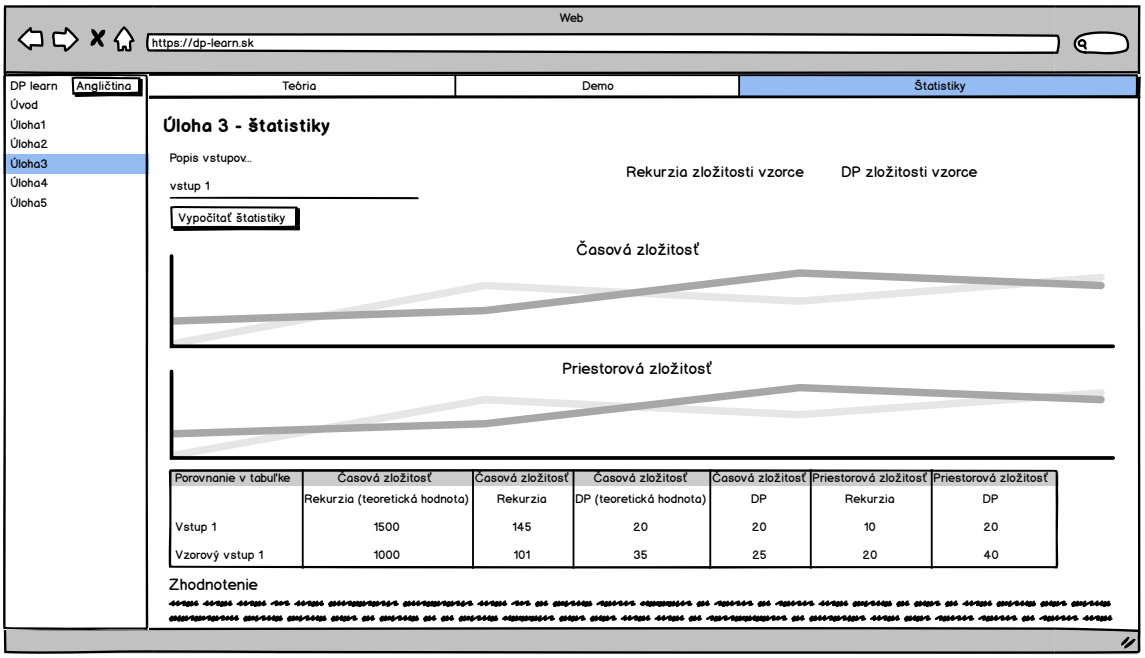
Záložka **Demo** obsahuje kľúčovú funkcionalita aplikácie - grafické znázornenie priebehu DP algoritmu. Vo vrchnej časti sa bude nachádzať formátovaný text. V ňom bude najprv popísané, v akom tvare treba zadať vstupné hodnoty, a aký je ich rozsah. Okrem toho bude v texte vysvetlený aj význam jednotlivých častí tabuľky. Ďalej sa budú vľavo nachádzať polia pre vstupné hodnoty a ovládacie prvky. Tlačidlom sa bude dať zvoliť rýchlosť priebehu algoritmu, pričom na výber bude aj možnosť ísť krok po kroku. Budú tu 3 tlačidlá – spustenie algoritmu, krokovanie a ukončenie algoritmu (okamžitý výpočet). Napravo bude formula, ktorá popisuje výpočet hodnôt do buniek tabuľky. Po zadaní vstupu bude možné spustiť DP algoritmus. Zobrazí sa tabuľka, v ktorej záhlaví sa nachádza popis aktuálnej operácie. V priebehu výpočtu budú farebne vyznačené bunky s ktorými algoritmus momentálne pracuje. Po vyplnení tabuľky (dokončení výpočtu) sa nad tabuľkou zobrazí výsledok. Mockup záložky je na obrázku 3.6.



Obr. 3.6: Mockup aplikácie pri otvorenej záložke `Demo`.

Vrchná časť záložky **Štatistiky** je podobná ako na záložke `Demo`. Namiesto formuly sú napravo uvedené zložitosti algoritmov. Po zadaní vstupu bude možné kliknutím na tlačidlo vykresliť grafy a tabuľku, ktoré porovnávajú zložitosť rekurzívneho a DP algoritmu. Pod tabuľkou bude formátovaný text so zhodnotením štatistík. Mockup záložky je na obrázku 3.7. Mockupy boli vytvorené pomocou webovej aplikácie **balsamiq**[9].

[9] http://balsamiq.cloud



Obr. 3.7: Mockup aplikácie pri otvorenej záložke `Štatistiky`.

3.4 Výber technológií pre vývoj

Keďže cieľom tejto práce je vytvoriť rýchlu, spoľahlivú a užívateľsky prívetivú aplikáciu, je potrebné si zvoliť vhodné technológie pre vývoj. V súčasnosti existuje veľké množstvo aplikačných rámcov a knižníc, ktoré veľmi uľahčujú a spríjemňujú vývoj webových aplikácií. Jednou z nich je pomerne nová knižnica s názvom **React**[10], na ktorej bude postavená táto aplikácia. Je to pomerne nová knižnica pre vývoj užívateľských rozhraní v jazyku JavaScript od firmy Facebook. V roku 2013, kedy ju Facebook zverejnil, priniesla zásadnú zmenu v tvorbe webových stránok. Kód napísaný v Reacte udáva, ako majú jednotlivé časti stránky vyzerať. Nieje potrebné písať zložitý kód, ktorý niečo mení. React automaticky vytvára virtuálny *DOM*[11], ktorý sa porovnáva so skutočným. Netreba kontrolovať aktuálny obsah. Ak prišlo k zmene DOM-u, stránka sa efektívne aktualizuje [7]. Ďalším špecifikom je, že React umožňuje kombinovať HTML, CSS a JavaScript v jednom zdrojovom súbore. Práve skladaním funkcií (alebo tried) v jazyku JavaScript vlastne definujeme HTML štruktúru. Tieto funkcie sa nazývajú *komponenty* [28]. Komponent sa automaticky prekresľuje na základe zmeny 2 vlastností [26]:

- *props* – vstupné parametre predávané komponentu
- *state* – špeciálny objekt, ktorý obsahuje rôzne premenné. Zmena objektu sa vykonáva zavolaním špeciálnej metódy **setState()**.

Dôkazom toho, že sa React momentálne teší vysokej popularite je veľké množstvo komponentov, ktoré sú voľne dostupné ako **npm** balíčky[12].

[10] https://reactjs.org/
[11] https://www.w3schools.com/whatis/whatis_htmldom.asp
[12] https://docs.npmjs.com/about-packages-and-modules

Okrem knižnice React som pri výbere technológií premýšľal nad ďalšími dvoma alternatívami. **Angular** je robustný aplikačný rámec na tvorbu webových, mobilných aj desktopových aplikácií [1]. V súčasnosti sa používa jeho druhá verzia, jeho popularita však postupne klesá. **Vue.js** je progresívny, vysoko škálovateľný aplikačný rámec, ktorý naopak naberá na popularite. Jeho výhodou je, že v základnej verzii sa sústreďuje výhradne na zobrazovanie informácií (rovnako ako React), ale dá sa kombinovať s ďalšími nástrojmi a podpornými knižnicami [34]. Vo výsledku sa tak dá dosiahnuť komplexné riešenie, rovnako ako v prípade rámca Angular.

React som si vybral z toho dôvodu, že ide o knižnicu, ktorá sa stará len o zobrazovanie informácií a nekladie presné požiadavky, ako písať aplikačnú logiku (čo sa úplne nedá povedať o rámci Angular). Oproti Vue.js zaváži fakt, že React má zatiaľ väčšiu užívateľskú základňu, čo znamená aj väčší počet podporných knižníc. Výber technológií som navyše konzultoval s ľuďmi z praxe, ktorí mi React odporučili.

Material-UI[13] je, rovnako ako React, open-source knižnica. Obsahuje React komponenty, ktoré implementujú tzv. *Material Design*. Material Design je dizajnový jazyk, ktorý predstavila firma Google v roku 2014, aby umožnila návrh jednotného dizajnu naprieč všetkými aplikáciami (mobilné, webové ale aj desktopové) [18]. Využitie tejto knižnice zabezpečí konzistentný dizajn celej aplikácie a ušetrí čas potrebný na definovanie vlastných štýlov, komponentov atď. Prostredie navyše vyzerá moderne a užívateľ tento štýl pozná z iných aplikácií. Tým pádom sa bude v aplikácii vedieť jednoduchšie orientovať a bude sa mu s ňou príjemne pracovať.

Aj keď je JavaScript stále jedným z najpopulárnejších jazykov nielen pre vývoj webových aplikácií, má svoje slabé stránky. Absencia tried (JavaScript pozná iba objekty), rozhraní a hlavne slabá typová kontrola prinášajú komplikácie pri vývoji komplexnejších aplikácií. Do popredia sa preto stále viac tlačia jeho „príbuzné“ jazyky, ktoré sú síce vo výsledku prekladané do jazyka JavaScript, ale ich kód je prehľadnejší a bezpečnejší, aj keď niekedy o niečo dlhší. Medzi najpopulárnejšie patria TypeScript (v ktorom je napísaná aj výsledná aplikácia), CoffeeScript alebo Dart.

TypeScript[14] je staticky typovaný, objektovo-orientovaný jazyk. Ide o nadmnožinu jazyka JavaScript, takže každý kód v jazyku JavaScript je aj korektným kódom v jazyku TypeScript. Jeho kód je *„transpilovaný“* do jazyka JavaScript, čo znamená že výsledkom prekladu je práve kód v jazyku JavaScript [16]. Už z názvu tohto jazyka vyplýva, že bude navyše obsahovať typy, triedy a rozhrania. Tieto prvky zaistia, že veľký počet chýb bude odhalených už pri statickej analýze kódu. Vývojové nástroje vďaka typom navyše ponúkajú relevantnejšie návrhy na automatické dopĺňanie častí kódu. Keďže knižnica React a aj väčšina podporných knižníc majú svoju typovanú verziu, použitiu jazyka TypeScript teda nič nebráni.

Visual Studio Code[15] je textový editor zameraný na písanie zdrojového kódu. Vo svojej základnej verzii obsahuje zvýrazňovanie syntaxe, inteligentné dopĺňanie kódu a možnosti refaktorizácie pre najpoužívanejšie programovacie jazyky. Dostupný je aj veľký počet balíčkov, ktoré pridávajú podporu aj pre menej známe jazyky, a pridávajú rôzne funkcie, vďaka ktorým sa tento textový editor stáva prakticky plnohodnotným, ale pritom jednoduchým vývojovým prostredím. V súčastnosti patrí medzi najpoužívanejšie prostredia pri vývoji webových aplikácií.

[13] https://material-ui.com/
[14] https://www.typescriptlang.org/
[15] https://code.visualstudio.com/

Zhrnutie vybraných technológií:

- **React** – knižnica, na ktorej bude postavená aplikácia
- **Material-UI** – design aplikácie
- **TypeScript** – programovací jazyk
- **Visual Studio Code** – vývojové prostredie

Kapitola 4

Implementácia

V tejto kapitole je podrobne rozobraná implementácia webovej aplikácie **DP learn**, ktorá bola prevedená podľa návrhu z predošlej kapitoly. Prvá časť sa venuje logickému rozdeleniu zdrojových súborov do adresárovej štruktúry. Nasleduje popis hlavných ovládacích prvkov aplikácie. V samostatných častiach sú potom rozobrané záložky pri optimalizačných úlohách – `Teória`, `Demo` a `Štatistiky`. Popísané sú zaujímavé riešenia z hľadiska implementácie. Ak je rovnaké riešenie použité na viacerých miestach, bude spomenuté iba raz.

4.1 Štruktúra zdrojových textov

Základom každej aplikácie postavenej na knižnici React sú už spomínané komponenty. Jeden React komponent v jazyku TypeScript predstavuje jednu triedu. Každá trieda sa nachádza v samostatnom súbore. Aj pri jednoduchej aplikácii sa teda dá očakávať vyšší počet zdrojový súborov. Pri komplexnej aplikácii je teda nevyhnutné vhodne rozdeliť zdrojové súbory do logických celkov na základe funkcionality, ktorú poskytujú. Vývojové prostredia vedia síce automaticky importovať komponenty z iných súborov, ale ak by boli všetky zdrojové súbory v jednom adresári, programátor by sa pri vývoji strácal. Pri logickom rozdelení sa aj ďalší programátor, ktorý sa k vývoju pripojí neskôr, v projekte zorientuje rýchlejšie.

V koreňovom adresári projektu sa nachádzajú rôzne konfiguračné súbory. Ide napríklad o nastavenie príznakov pre prekladač, zoznam použitých knižníc, nastavenie publikovania aplikácie atď.

V adresári **public** sa nachádzajú súbory, úpravou ktorých sa dá napríklad zmeniť titulok alebo ikona webovej stránky. Všetky ostatné zdrojové súbory sa nachádzajú v adresári **src** a sú rozdelené nasledovne:

- **components** – súbor `App.tsx` obsahuje jednoduchý komponent, ktorý je „koreňom“ aplikácie. `Page.tsx` predstavuje jedinú stránku aplikácie, ktorá využíva všetky ostatné komponenty. Ide teda o tzv. *single-page* aplikáciu (jediná stránka, ktorá sa skladá z komponentov, a tie sa prekresľujú samostatne na základe interakcie s užívateľom [15]). Ďalšie komponenty sú rozdelené podľa ich vlastností do ďalších podadresárov:
  - **content** – komponenty ktoré sú v aplikácii nositeľmi obsahu (úvodná stránka a záložky optimalizačných úloh)
  - **customStyled** – Material-UI komponenty s mierne upraveným štýlom, napr. farba alebo veľkosť

  - **hoc** – *higher-order component* (v preklade komponent vyššieho rádu) je funkcia, ktorá berie ako parameter komponent a vracia iný komponent
  - **specialized** – komponenty, ktoré sú svojou funkcionalitou špecifické pre aplikáciu. Ide napríklad o zobrazenie tabuľky v záložke `Demo`, zobrazenie zdrojových kódov, vykreslenie grafov atď.
- **resources** – všetky obrázky a ikony použité v aplikácii
- **statsHelpers** – súbory v tomto adresári obsahujú funkcie, ktoré sú používané pri výpočte štatistík v záložke `Štatistiky`
- **strings** – nachádzajú sa tu reťazcové konštanty v 2 podadresároch:
  - **dpProblemsStrings** – zdrojové kódy v jazyku C a zložitosti rekurzívnych a DP algoritmov jednotlivých optimalizačných úloh
  - **translations** – texty použité v aplikácii
- **styles** – definície štýlov

Priamo v adresári **src** sa ešte nachádzajú tieto 3 súbory:

- `registerServiceWorker.ts` – skript spúšťajúci server, na ktorom beží aplikácia
- `index.tsx` – hlavný skript každej React aplikácie, ktorý volá koreňový **App** komponent
- `helpers.tsx` – súbor obsahuje pomocné funkcie, ktoré sú používané na rôznych miestach v aplikácii

4.2 Hlavné ovládacie prvky aplikácie

Aplikácia je rozdelená na 2 časti – ovládacie prvky na ľavej strane a hlavné okno s obsahom, ktoré zaberá zvyšok obrazovky. V ľavom hornom rohu sa vedľa názvu aplikácie nachádza tlačidlo na prepínanie jazyka aplikácie (anglický a slovenský). Túto časť predstavuje komponent **MenuHeader**. Tlačidlom je Material-UI **Button**, v ktorom sa nachádza HTML element **<img>**. Ikony pochádzajú zo stránky **Pixabay**[1], pričom sú bezplatné aj pre komerčné využitie. Tlačidlo sa teda zobrazí ako vlajka podľa aktuálne zvoleného jazyka. Táto informácia sa zisťuje metódou **getLanguage()**. Po kliknutí na tlačidlo sa prepne jazyk aplikácie. K tomu slúži zasa metóda **setLanguage()**. Obe metódy pochádzajú z knižnice **react-localization**[2], ktorá zabezpečuje jazykovú lokalizáciu aplikácie. Jazyky sú definované v súbore `strings.tsx`. Konkrétne reťazce sú uložené v samostatných súboroch. Každý súbor obsahuje jeden objekt, v ktorom sú reťazce. Vo výpisoch 4.1 a 4.2 je zobrazené, ako vyzerá obsah týchto súborov.

[1] https://pixabay.com
[2] https://www.npmjs.com/package/react-localization

```
export const english = {
    global: {
        ...
    },
    demoGlobal: {
        assigning: 'Assigning ',
        loop: 'Loop',
        start: 'Start',
        array: 'Array',
        ...
    },
    ...
}
```

Výpis 4.1: Časť súboru english.tsx.

```
export const slovak = {
    global: {
        ...
    },
    demoGlobal: {
        assigning: 'Priradzujem',
        loop: 'Cyklus',
        start: 'Start',
        array: 'Pole',
        ...
    },
    ...
}
```

Výpis 4.2: Časť súboru slovak.tsx.

Aj v prípade, že sa reťazce v oboch jazykoch zhodujú, sú obsiahnuté v týchto objektoch. Je to z dôvodu dodržania konzistencie. Na základe aktuálne zvoleného jazyka sa v aplikácii zobrazujú reťazce z jedného alebo druhého objektu.

Menu aplikácie je implementované pomocou Material-UI komponentu **List** s položkami **ListItem**. Položky sú špeciálne naštýlované, aby bolo zjavné, ktorá položka je aktuálne zvolená. Po kliknutí na položku sa zmení obsah hlavného okna aplikácie. Využíva sa k tomu tzv. *Conditional Rendering*, ktorý umožňuje zobraziť alebo nezobraziť určitý komponent, na základe toho, či bola alebo nebola splnená podmienka [27].

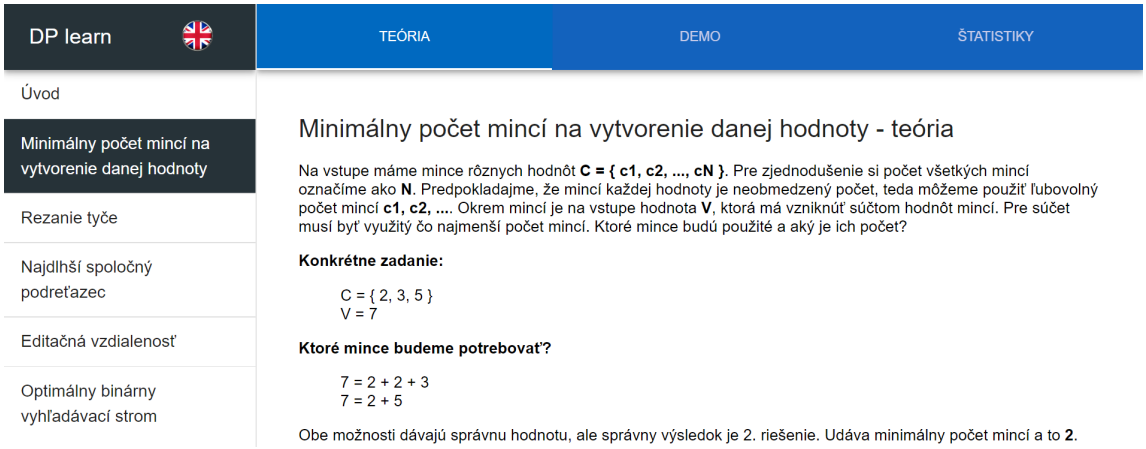
Ak je v menu zvolená niektorá optimalizačná úloha, hlavné okno aplikácie obsahuje 3 záložky, medzi ktorými sa dá prepínať. Túto funkcionalitu zabezpečuje Material-UI komponent **Tabs**, ktorý združuje záložky **Tab** a riadi ich prepínanie.

Aplikácia využíva *cookie*[3] na ukladanie určitých stavov. Je to z toho dôvodu, aby mohol užívateľ po znovuotvorení stránky pokračovať tam, kde prestal. Do cookie sa ukladá stav práve spomenutých ovládacích prvkov aplikácie. Ide teda o zvolený jazyk, položku v menu a otvorenú záložku pri optimalizačnej úlohe. S cookie sa dá jednoducho pracovať pomocou knižnice **universal-cookie**[4]. Na mieste, kde treba zapisovať alebo čítať z cookie, stačí vytvoriť objekt typu **Cookies**. Okrem iného sú potom k dispozícii metódy **get()** (získanie hodnoty cookie) a **set()** (nastavenie hodnoty cookie). Ak sa podarí získať hodnotu cookie, aplikácia prejde do daného stavu. V opačnom prípade sa nastaví predvolená hodnota (slovenský jazyk, prvá položka z menu a záložka s teóriou).

K dosiahnutiu výsledného zobrazenia komponentov ako na obrázku 4.1 je potrebné definovať štýly. React ponúka viacero možností. Pri implementácii boli využité 2 z nich. Klasické CSS v súbore `index.css` definuje štýl okrajov a písma pre element **<body>**. Štýl pre jednotlivé elementy a komponenty je definovaný pomocou *JSS*, ktoré umožňuje písať CSS štýly v jazyku JavaScript [17]. CSS súbor je vytvorený až pri kompilácii. Má to niekoľko výhod. Užívateľ môže vidieť výhodu vo výkonnosti - CSS súbor sa vytvára až za behu, na základe vykreslenia komponentu, ktorý definíciu štýlu používa. Programátor využitie JSS ocení hlavne pri vývoji. Vďaka statickej analýze dokáže vývojové prostredie detekovať preklepy (použitá trieda štýlu, ktorá neexistuje), automaticky navrhovať kód atď.

JSS štýly sú definované väčšinou v tom súbore, kde je aj kód komponentu. Výnimkou je súbor `globalStyles.tsx`. Ten obsahuje definíciu štýlov, ktoré sú využívané vo viacerých

[3] https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies
[4] https://www.npmjs.com/package/universal-cookie



Obr. 4.1: Hlavné ovládacie prvky aplikácie – prepínač jazykov, menu a záložky pri otvorenej optimalizačnej úlohe.

komponentoch. Ak je rovnaká trieda štýlu používaná na viacerých miestach v aplikácii, nachádza sa v tomto súbore, aby nemusela byť definovaná pre každý komponent zvlášť. Ide hlavne o štýlovanie tabuľky, ktoré prebieha v komponentoch záložky `Demo` a v komponente **StatsTable**. Niektoré triedy sú používané zasa v komponentoch záložky `Teória`. Okrem štýlov sa v tomto súbore nachádza aj definícia farieb používaných v aplikácii.

4.3 Úvodná stránka

Úvodná stránka je dostupná po kliknutí na prvú položku v menu. Pre zobrazenie nadpisov a podnadpisov je využitý komponent **CustomTitle**. Oproti Material-UI komponentu **Typography** naviac obsahuje vrchné a spodné odsadenie.

Informácie o aplikácii sú zobrazené ako formátovaný text pomocou knižnice **react-markdown**[5]. Reťazec musí byť zapísaný v značkovacom jazyku **Markdown**[6]. Toto riešenie umožňuje použiť zároveň formátovaný text a viaceré jazykové varianty aplikácie.

Zoznam vlastností, ktoré musí rekurzívne riešenie úlohy spĺňať, je zobrazený jednoducho pomocou HTML elementov **<ul>** a **<li>**.

Popis jednotlivých záložiek pri optimalizačných úlohách je vložený do komponentu **LeftMarginDiv**, ktorý pridáva obsahu odsadenie od ľavého okraju.

Položky v referenciách sú klikateľné odkazy (HTML element **<a>**) a zároveň sú tučným písmom. Na dosiahnutie tohto stavu bolo treba miesto značkovacieho jazyka použiť HTML element **<b>**.

4.4 Záložka s teóriou k optimalizačnej úlohe

Úvodná časť s príkladom a jeho riešením je vložená do komponentu **BottomMarginDiv**, ktorý pridáva spodné odsadenie. Navyše je táto časť oddelená aj HTML elementom **<hr>** (vodorovná čiara).

[5] https://www.npmjs.com/package/react-markdown
[6] https://www.markdownguide.org/

V prostrednej časti je vedľa seba uvedené rekurzívne a DP riešenie úlohy. Rovnomerné rozdelenie stránky zabezpečí kód zobrazený vo výpise 4.3.

```
<Grid container={true} direction='row'>
    <FlexOne>
        ...
    </FlexOne>
    <FlexOne>
        ...
    </FlexOne>
</Grid>
```

Výpis 4.3: Kód zabezpečujúci rovnomerné zobrazenie obsahu vedľa seba.

Komponent **FlexOne** má v CSS nastavenú hodnotu `flex: 1`. V kombinácii s Material-UI komponentom **Grid** s danými parametrami je tak zabezpečené správne zobrazenie pomocou CSS zobrazovacieho modulu s názvom *Flexible Box*. V prípade, že je potrebné rozmiestniť niekoľko prvkov v určitom smere, ideálne je k tomu využiť práve tento modul [24].

Rámček s časovou a priestorovou zložitosťou zobrazuje komponent **Complexity**. Komponent berie ako parametre reťazce so zložitosťami a navyše po zadaní aj tretieho parametru môže zobraziť informáciu o aký typ algoritmu (rekurzívny alebo DP) ide. Parameter má definované 3 hodnoty, ktoré môže nadobúdať (`'rec' | 'dp' | undefined`) a na základe toho zobrazí alebo nezobrazí nadpis. Komponent má definované predvolené parametre v statickej premennej **defaultProps**, ktorá je špecifická pre React [4]. Všetky parametre sú teda nepovinné. V rámčeku sa zobrazia len tie informácie, ktoré boli poskytnuté. Ikony pre čas a priestor sú z knižnice Material-UI.

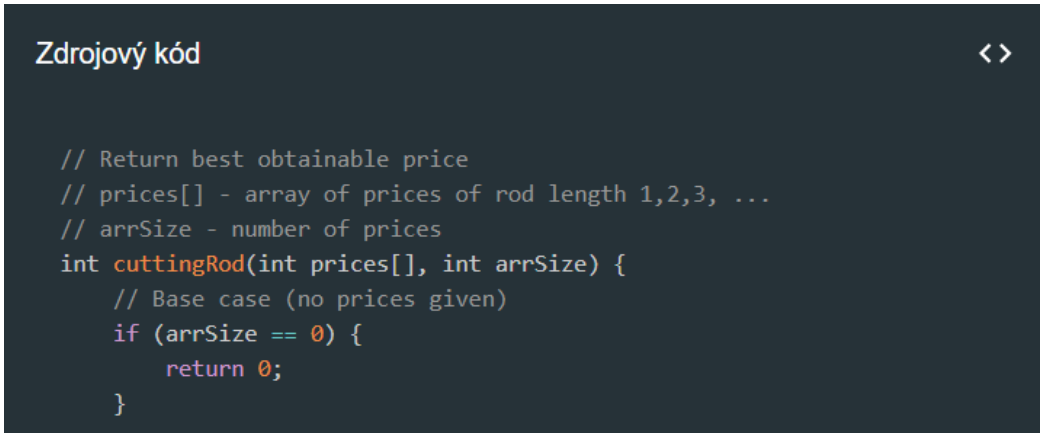
Strom rekurzívnych volaní je nakreslený v aplikácii *draw.io*[7] a exportovaný vo vektorovom formáte SVG (v rovnakej aplikácii sú nakreslené aj stromy uvedené v tejto správe). Tento obrázok vstupuje ako parameter do komponentu **PaddingImage**, ktorý mu pridáva odsadenie od okrajov.

Tabuľky, na ktorých je znázornený výber/výpočet novej hodnoty bunky, pochádzajú zo záložky `Demo`. Ak je to pri konkrétnej úlohe potrebné, šipky naznačujúce výber hodnôt sú dokreslené v rovnakej aplikácii ako stromy rekurzívnych volaní.

Zobrazenie zdrojových kódov zabezpečuje komponent **SourceCode**. Zdrojový kód vstupuje do komponentu ako reťazec, ktorý je vložený do HTML elementov **<code>** a **<pre>**. Zvýrazňovanie syntaxe jazyka C zapezpečuje *Prism*[8]. Štýl je definovaný v súbore `prism.css`. **SourceCode** je jediný komponent, v ktorom je využitá špeciálna React metóda **componentDidMount()**. Táto metóda je zavolaná po tom, ako sú vykreslené všetky komponenty [5]. V tejto chvíli už je možné zavolať Prism metódu **highlightAll()**, ktorá zvýrazní syntax zdrojových kódov.

Material-UI komponent **ExpansionPanel** a jeho „príbuzné“ komponenty umožňujú skryť a znovu zobraziť ľubovolný obsah. Po pridaní štýlu a Material-UI ikony **Code** vznikne tmavý otvárací panel so zdrojovým kódom (obrázok 4.2).

[7] https://www.draw.io/
[8] https://prismjs.com/



Obr. 4.2: Vykreslená časť komponentu **SourceCode**. Po kliknutí do vrchnej časti sa zdrojový kód skryje a zostane viditeľná len vrchná časť. Rovnakým spôsobom je možné zdrojový kód opäť zobraziť.

4.5 Záložka s grafickým zobrazením priebehu algoritmu

Vstupné hodnoty úlohy sú zadávané do komponentu **CustomTextField**, ktorého základom je Material-UI komponent **TextField**. Vstupnými parametrami sú `label` (zobrazený popis), `value` (zobrazený text) a `onChange()`. Posledný parameter je metóda, ktorá je volaná pri zmene textu. **TextField** zmenený text posiela tejto metóde v špeciálnom parametri. V aplikácii sú 3 typy hodnôt, ktoré môžu byť vstupom optimalizačnej úlohy:

- **reťazec** – v metóde **onChange()** je iba aktualizovaná premenná `value`
- **celé číslo** – pomocou pretypovania reťazca na číslo prebehne kontrola, či ide naozaj o celé číslo, a ak áno, premenná `value` je aktualizovaná
- **pole celých čísel** – funkcia **GetNumbers()** v `helpers.tsx` kontroluje, či reťazec obsahuje iba celé čísla oddelené čiarkami. Ak áno, vráti pole týchto čísel, pričom je aktualizovaná premenná `value`.

Navyše, pre každý vstup je definovaná maximálna hodnota. Môže to byť dĺžka reťazca, najvyššie možné číslo alebo počet čísel v poli. Hodnota je aktualizovaná iba ak vyhovuje podmienke. Každý `Demo` komponent má vlastné **onChange()** metódy.

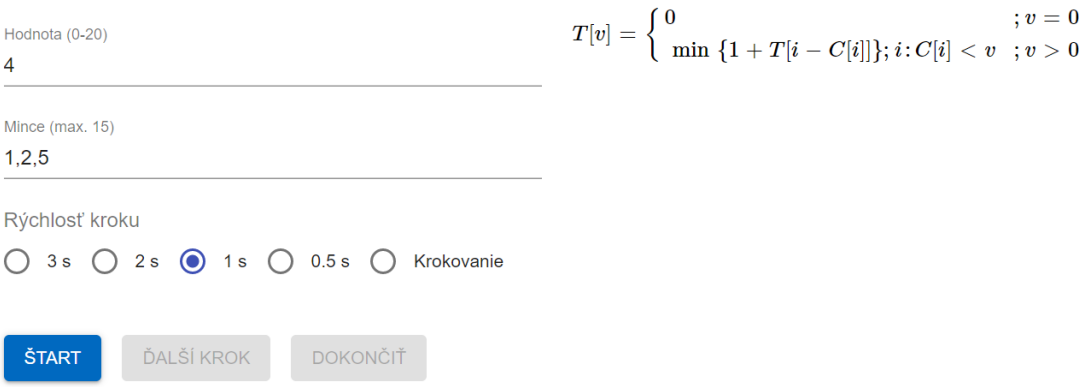
Komponent **SpeedSelector** sa stará o nastavenie rýchlosti priebehu algoritmu. Funguje podobne ako predchádzajúci komponent. Pri zmene rýchlosti je zavolaná metóda **onClick()**, ktorá aktualizuje hodnotu `speed`. Okrem toho, ak aktuálne prebieha algoritmus, vypína (ak je nastavená hodnota **0** - symbolizuje krokovanie) alebo zapína automatické vypĺňanie tabuľky.

Komponent **CustomButton** je použitý na tlačidlá pre spustenie, krokovanie a okamžité dokončenie algoritmu. Je to iba špeciálne naštýlovaný **Button** z knižnice Material-UI. Funkcie jednotlivých tlačidiel budú podrobne popísané neskôr.

Napravo od ovládacích prvkov sa nachádza formula popisujúca výber hodnôt do tabuľky. Formula je zapísaná pomocou značkovacieho jazyka *AsciiMath*[9]. Takýto popis formuly vstu-

[9] http://asciimath.org/

puje ako reťazec do komponentu **Formula**. Zobrazenie formuly podľa popisu v parametri zabezpečuje *MathJax*, konkrétne jeho 2. verzia[10]. Výsledné zobrazenie je na obrázku 4.3.



Obr. 4.3: Časť záložky `Demo` so vstupmi, ovládacími prvkami a formulou.

MathJax je jediná knižnica v aplikácii, ktorá nemá definíciu typov. Rýchle riešenie tejto situácie bolo vytvoriť zdrojový súbor `global.d.ts`. V tomto súbore stačí deklarovať danú knižnicu rovnako ako vo výpise 4.4.

```
declare module 'react-mathjax2';
```

Výpis 4.4: Deklarácia knižnice v súbore `global.d.ts`.

Rovnakým spôsobom by stačilo dopísať deklarácie ďalších knižníc v aplikácii, ktorým chýba definícia typov. Toto riešenie nie je ideálne, pretože bez definície typov nebudú prebiehať kontroly a môže nastať pád aplikácie za behu. Knižnica má však v aplikácii iba jedno jednoduché využitie, ktoré stačí otestovať, preto som sa rozhodol pre toto rýchle a jednoduché riešenie namiesto písania kompletného typovacieho súboru.

Po kliknutí na tlačidlo ŠTART je zavolaná metóda **handleStartClick()**. V tejto metóde sú nastavené všetky premenné a objekt `state` do predvoleného stavu – príprava DP algoritmu. Prebieha tu aj kontrola, či boli zadané validné vstupné údaje. Ak nie, algoritmus sa nespustí a do obdĺžnika sa vypíše chybová správa. Ak áno, vykoná sa prvý krok algoritmu. Ak nie je nastavené krokovanie, je zavolaná pomocná metóda **setTimeout()**, ktorá pomocou JavaScript funkcie s rovnakým názvom naplánuje volanie metódy v parametri na určitý čas podľa nastavenej rýchlosti vypĺňania tabuľky. Kód metódy **setTimeout()** je zobrazený vo výpise 4.5.

```
private setTimeout = (func: () => void) => {
    this.timeout = setTimeout(func, 9000 / this.state.speed);
}
```

Výpis 4.5: Metóda setTimeout() v `Demo` komponentoch.

Vykonávanie krokov algoritmu riadi metóda **finiteAutomata()**. Ide vlastne o konečný stavový automat, kde na základe stavu v premennej `nextAutomataState` je zavolaná konkrétna metóda, ktorá vykoná krok algoritmu. Ak nie je zvolené krokovanie, alebo automat nie je v stave `'done'`, naplánuje sa opätovné volanie metódy. V prípade krokovania je metóda **finiteAutomata()** volaná vždy pri kliknutí na tlačidlo ĎALŠÍ KROK. Po dokončení

[10] https://www.npmjs.com/package/react-mathjax2

algoritmu je nastavený stav automatu `'done'` a zavolaná metóda **setFinalState()**. Tá nastaví hodnoty v objekte `state` tak, aby signalizovali koniec algoritmu. Tým je zaručené, že sa správne zobrazia (alebo skryjú) niektoré komponenty, zruší sa vyznačenie buniek v tabuľke atď. Okrem toho tu prebieha zostavenie celého riešenia (okrem úlohy *Optimálny binárny vyhľadávací strom*).

Ak chce vidieť užívateľ vyplnenú tabuľku a výsledok hneď, stačí kliknúť na tlačidlo DOKONČIŤ. K tlačidlu je priradená metóda **handleFinishClick()**, ktorá vykoná celý výpočet okamžite.

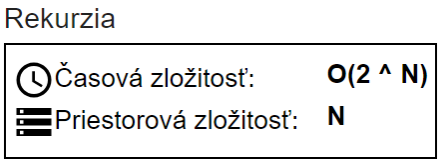
Tabuľku predstavuje komponent **DemoTable**. Používa Material-UI komponent **Table** a jeho „príbuzné“ komponenty. Okrem samotnej tabuľky komponent zobrazuje aj výsledok úlohy. Súčasťou tabuľky je aj popis aktuálneho kroku algoritmu. Obsah tabuľky je predávaný v parametroch komponentu. Metódy **tableHead()** a **tableBody()** sa nachádzajú priamo v `Demo` komponentoch a definujú obsah a vzhľad tabuľky na základe aktuálnych hodnôt premenných a objektu `state`, ktoré sú nastavované v priebehu algoritmu. V bunkách sa zobrazuje buď číslo, súčet čísel alebo špeciálny reťazec. Funkcie **ValueOrIntMax()** a **ValueOrUndefined()** zo súboru `helpers.tsx` berú ako parameter číslo a vrátia buď reťazec rovný hodnote alebo špeciálny reťazec, ak je v parametri špeciálna hodnota (`'INT_MAX'` alebo `'-'`). Tabuľka v priebehu výpočtu je na obrázku 4.4.

Priradzujem 10, použitá dĺžka 2						
	Dĺžka 0	Dĺžka 1	Dĺžka 2	Dĺžka 3	Dĺžka 4	Dĺžka 5
Najvyššia cena	0 + 6	1 + 6	5 + 5	6 + 1	10	-
Použitá dĺžka	-	1	2	1	2	-

Obr. 4.4: Tabuľka zobrazujúca priebeh DP algoritmu pri úlohe *Rezanie tyče*.

4.6 Záložka s porovnaním rekurzívneho a DP algoritmu - štatistiky

Časová a priestorová zložitosť je zobrazená pomocou komponentu **Complexity**. Keďže je vedľa seba zobrazená zložitosť rekurzívneho aj DP algoritmu, je treba vyjadriť, ku ktorému algoritmu daná zložitosť patrí. To je docielené zadaním všetkých troch parametrov komponentu. V takom prípade sa zobrazuje aj popis, o aký typ algoritmu ide (obrázok 4.5).



Obr. 4.5: Vykreslený komponent **Complexity** so zadanými všetkými vstupnými parametrami, pričom `recOrDp == 'rec'`.

Po kliknutí na tlačidlo VYPOČÍTAŤ ŠTATISTIKY je zavolaná metóda **drawStats()**. Tá najprv vynuluje premenné so štatistikami, prevedie vstupné hodnoty na správny typ a zavolá metódu **getStats()**. Táto metóda sa stará o naplnenie premenných so štatistikami. V adresári **statsHelpers** sa nachádza pre každú optimalizačnú úlohu 1 súbor. Tento súbor obsahuje funkcie na vypočítavanie zložitosti rekurzívneho a DP algoritmu pre vstupné hodnoty zadané užívateľom. Okrem toho sa tu nachádza konštanta, ktorá obsahuje pole vzorových vstupných hodnôt a štatistík, ktoré k vstupným hodnotám náležia.

Priestorová zložitosť je vypočítaná jednoducho podľa vzorca na základe vstupných hodnôt, rovnako ako aj maximálna teoretická hodnota **časovej zložitosti**. Pri DP algoritme je navyše vypočítaný presný počet behov cyklu a pri rekurzívnom algoritme počet volaní funkcie. K tomu je potrebné vykonať kompletný kód daného algoritmu. Aby bolo možné zrátať počet rekurzívnych volaní, je ako parameter využitý objekt typu **ISimpleObjectParameter** (výpis 4.6), ktorý je definovaný v `CoinsStatsHelper.tsx`.

```
export interface ISimpleObjectParameter {
    value: number
}
```

Výpis 4.6: Objekt používaný na počítanie rekurzívnych volaní funkcie.

Využitie objektu je nutné z toho dôvodu, že objekty ako parametre metódy sú predávané odkazom. Tým pádom sa pri inkrementovaní `value` zmení hodnota v pôvodnom objekte, nie len lokálne v metóde. Funkcia je vždy volaná s objektom, kde `value == 0`. Pri rekurzívnom algoritme je hodnota `value` inkrementovaná vždy na začiatkufunkcie. Na konci výpočtu sa bude hodnota `value` rovnať počtu volaní funkcie (počíta sa teda aj prvé volanie funkcie).

Pri DP algoritme sa hodnota inkrementuje v najhlbšie zanorenom cykle, ak je vykonaná nejaká operácia. Napríklad pri úlohe *Minimálny počet mincí* sa hodnota inkrementuje iba ak je splnená podmienka vo vnútornom cykle - preto sa môže časová zložitosť líšiť od teoretickej hodnoty aj pri DP algoritme. Na konci výpočtu sa hodnota `value` rovná počtu vykonaní najhlbšie zanoreného cyklu.

Vypočítané hodnoty sú priradené do týchto premenných:

- **spaceChartStats** – pole hodnôt priestorovej zložitosti (vstupný parameter komponentu **SpaceGraph**)
- **timeChartStats** – pole hodnôt časovej zložitosti (vstupný parameter komponentu **TimeGraph**)
- **tableStats** – pole hodnôt oboch zložitostí (vstupný parameter komponentu **StatsTable**)

Okrem hodnôt zložitostí tieto objekty obsahujú vždy aj reťazec – vstupné hodnoty, ku ktorým vypočítané hodnoty patria.

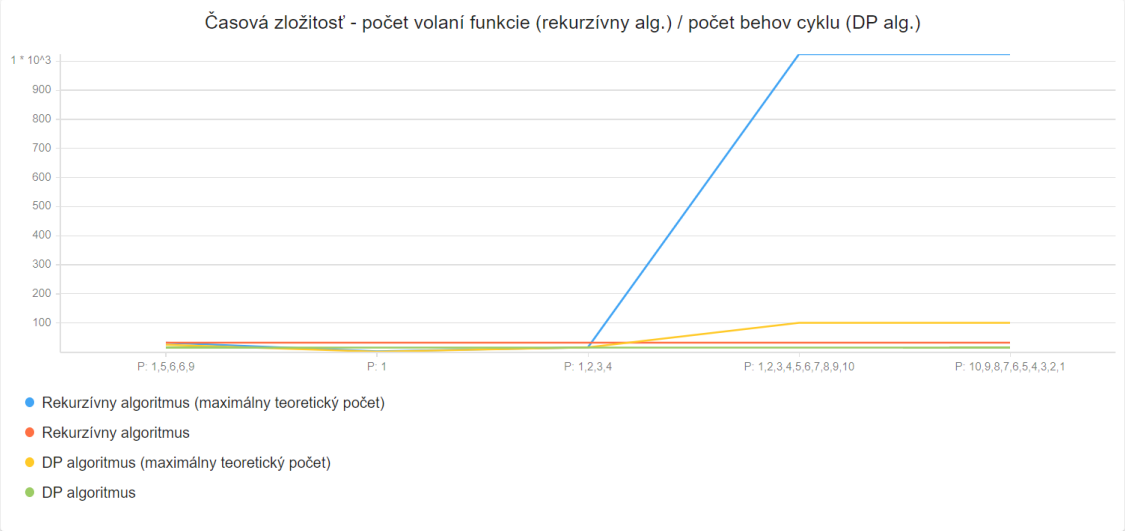
Vykresľovanie grafov zabezpečuje knižnica **dx-react-chart**[11]. Práca s knižnicou je veľmi jednoduchá. Do komponentu **Chart** stačí pridávať ďalšie komponenty z knižnice na základe toho aké dáta sa majú zobraziť a ako má graf vyzerať.

[11] https://devexpress.github.io/devextreme-reactive/react/chart/docs/guides/getting-started/

V aplikácii boli použité tieto komponenty:

- **ValueScale** – určenie hodnoty, ktorá slúži ako mierka grafu
- **ArgumentAxis**, **ValueAxis** – zobrazenie osí grafu. Metóda **getLabel()** v komponente **TimeChart** vracia popis hodnoty na ose y (**ValueAxis**), pričom ide len o prevod čísla do tvaru $x * 10 \hat{} n$.
- **LineSeries** – pridanie čiary do grafu. Hodnota z objektu s dátami, ktorú má čiara znázorňovať je definovaná v parametroch, podobne ako farba, názov atď.
- **Legend**, **Animation** – zobrazenie legendy a zapnutie animácií pri vykreslení

Všetko je spojené do komponentov **TimeChart** a **SpaceChart**, ktoré majú jediný vstupný parameter – dáta k zobrazeniu. Zobrazujú aj nadpis, ktorý udáva informáciu, o aké štatistiky ide. Takto vykreslený graf je na obrázku 4.6.



Obr. 4.6: Vykreslený komponent **TimeChart**.

Okrem grafov sú všetky vypočítané hodnoty zhrnuté v jednej tabuľke – komponent **StatsTable**. Na rozdiel od komponentu **DemoTable** má tento komponent jediný vstupný parameter, a to dáta k zobrazeniu. Metódy, ktoré definujú vzhľad a obsah tabuľky na základe vstupných dát sa nachádzajú priamo v komponente.

Kapitola 5

Testovanie aplikácie a experimenty

Táto kapitola je rozdelená na tri časti. V prvej sa venuje testovaniu aplikácie z hľadiska funkčnosti, spoľahlivosti a užívateľskej prívetivosti. V druhej časti je popis úprav, ktoré boli v aplikácii prevedené na základe výsledkov testovania. Posledná časť sa venuje experimentom prevedeným na konkrétnych optimalizačných úlohách pri využití rekurzívneho aj DP algoritmu.

5.1 Testovanie aplikácie

Okrem samotnej implementácie je veľmi dôležitý proces testovania. Dynamické časti aplikácie boli v priebehu vývoja manuálne testované vždy na základnej množine vstupov. Veľa chýb tak bolo odhalených už pri vývoji. Po dokončení vývoja bola opäť otestovaná celková funkcionalita a tiež rôzne rozlíšenia obrazovky, aby bola aplikácia použiteľná aj na starších monitoroch.

Po tomto testovaní bola aplikácia nasadená na verejne prístupný server, aby mohlo prebehnúť aj užívateľské testovanie. K tomu bola využitá bezplatná služba **Firebase Hosting**, ktorá poskytuje rýchly a bezpečný hosting webových aplikácií [8]. Užívateľské testovanie prebiehalo formou dotazníka, ktorý bol vytvorený pomocou služby **Survio**[1].

Dotazník vypĺňali rôzne skupiny užívateľov. Išlo o študentov informatiky, skúsených softvérových vývojárov ale aj laikov v oblasti informačných technológií. Dôvodom bolo, že aj bežný užívateľ môže poskytnúť cenné pripomienky, ktoré môžu byť užitočné pri ladení užívateľského prostredia. Testovania sa zúčastnilo dokopy **21** ľudí, z ktorých vyše 70% uviedlo, že sa pohybuje v oblasti informačných technológií. Pre zaujímavosť bola položená aj otázka, či sa užívateľ stretol s dynamickým programovaním. Zaujímavé je, že ani niektorí užívatelia, ktorí sa pohybujú v oblasti informačných technológií sa s touto technikou nestretli, čo dokazuje graf v prílohe A.1.

Testovací scenár sa nachádzal priamo v dotazníku. Pred každou skupinou otázok bolo vždy popísané, aké akcie má užívateľ vykonať. Na začiatku si mal užívateľ zvoliť jazyk aplikácie a jednu z optimalizačných úloh, s ktorou bude pracovať v priebehu testovania. Testovanie kompletne celej aplikácie by užívateľovi zabralo príliš veľa času.

Keďže si až tretina zo všetkých užívateľov vybrala anglický jazyk, dá sa považovať pridanie tejto jazykovej varianty za užitočné. Najčastejšie bola volená úloha *Minimálny počet mincí* a naopak úlohu *Optimálny binárny vyhľadávací strom* si nevybral žiadny užívateľ.

[1] https://www.survio.com/

Prejavilo sa teda poradie úloh v menu. Užívateľ mal nasledovne otestovať jednotlivé záložky podľa pripraveného scenára:

1. **Teória**
   Užívateľ mal za úlohu preskúmať záložku, zhodnotiť rozdelenie informácií do jednotlivých blokov a napísať, ak mu chýbali v záložke nejaké informácie.

2. **Demo**
   Po zadaní vstupných hodnôt mal užívateľ sledovať priebeh algoritmu pri rôznych rýchlostiach, vyskúšať si krokovanie, a tiež možnosť okamžitého vyplnenia tabuľky. Následne mal zhodnotiť ako mu vyhovovalo takéto zobrazenie priebehu algoritmu, a zaznamenať prípadné problémy.

3. **Štatistiky**
   Opäť bolo treba zadať vstupné hodnoty a spustiť výpočet štatistík. Užívateľ mal následne zhodnotiť, či mu prišlo zobrazenie štatistík prehľadné, či mu nechýbalo porovnanie algoritmov ešte z iného hľadiska a zaznamenať prípadné problémy.

V tejto časti mali užívatelia možnosť slovného hodnotenia, na základe ktorého boli v aplikácii prevedených niekoľko úprav. Odpovede aj úpravy budú uvedené v ďalšej časti.

V závere dotazníka boli položené 3 otázky, ktoré ponúkli priestor na celkové zhodnotenie aplikácie:

1. **Ako hodnotíte celkový vzhľad aplikácie?**
   Viac ako 60% užívateľov ohodnotilo vzhľad aplikácie ako „veľmi dobrý“, 33% ako „priemerný“ a jeden užívateľ ho ohodnotil ako „podpriemerný“. Výsledný graf sa nachádza v prílohe A.2.

2. **Aká náročná je orientácia v aplikácii?**
   Takmer rovnakým dielom boli rozdelené odpovede „veľmi jednoduchá“ a „skôr jednoduchá“. Zvyšní dvaja užívatelia hodnotili orientáciu v aplikácii ako „priemernú“. Výsledný graf sa nachádza v prílohe A.3.

3. **Odporučili by ste aplikáciu ako vhodný prostriedok na podporu výučby dynamického programovania?**
   Aj pri tejto otázke prevažovali pozitívne odpovede. 57% užívateľov odpovedalo „áno“, 38% „skôr áno“ a jeden užívateľ odpovedal „neviem“. Výsledný graf sa nachádza v prílohe A.4.

Aj keď u prvých dvoch otázok prevládajú pozitíve odpovede, určite sa dá v aplikácii nájsť priestor na vylepšenia, preto boli na základe výsledkov testovania prevedené určité zmeny.

Podľa odpovedí na poslednú otázku sa dá povedať, že bol splnený cieľ práce, a to vytvoriť aplikáciu na podporu výuky dynamického programovania.

5.2 Úpravy aplikácie prevedené na základe výsledkov testovania

Po zhodnotení slovných odpovedí v dotazníku bolo v aplikácii prevedených niekoľko opráv a úprav. Najmä podnety od užívateľov pohybujúcich sa v oblasti informačných technológií boli naozaj relevantné, a aj vďaka nim môže byť aplikácia lepšou a prospešnejšou.

Zoznam prevedených opráv:

- Ošetrenie vstupných hodnôt a pridanie chybovej hlášky v prípade, že sa úloha nedá vyriešiť.

  > „*Aplikácia nefunguje s použitím mincí s desatinnou hodnotou. Tiež nefunguje pokiaľ sa požadovaná hodnota nedá zostaviť so zadaných mincí, aplikácia by mala vypísať chybovú hlášku.*“

- Nastavenie maximálnej šírky hlavného okna aplikácie a automatické zobrazenie *scroll baru*, ak je demo tabuľka širšia ako hlavné okno.

  > „*Jedna vec čo mi ale prekáža je pri 20 hodnotách extrémne široký kontajner tej tabuľky, ktorý rozšíri celú stránku, spolu s ostatnými časťami, ktoré sa nepríjemne roztiahnu. Nastavil by som na kontajner stránky na* `overflow: hidden` *a povolil „scroll“ len vrámci tabuľky, prípadne ju zalomil 2x10.*“

- Úprava popisov v grafe na osi X, pretože sa prekrývali. Dôvodom bolo pravdepodobne zadanie dlhého vstupu pri malom rozlíšení obrazovky.

  > „*Pri grafoch sa texty na spodnej osi prekrývajú a preto nie sú moc čitateľné.*“

- Oprava prepínača rýchlosti. Pri stlačení tlačidla bol nastavený vždy nový timeout, pričom pôvodný timeout nebol zmazaný. To, že bol nastavený viacnásobný timeout budilo dojem zvýšenia rýchlosti a nastával problém, že sa po kliknutí na tlačidlo DOKONČIŤ vykonalo niekoľko krokov navyše.

  > „*Pokud jsem provedl změnu animace několikrát rychle po sobě, jakoby se rychlosti sečetly. Navíc text popisující aktuální krok (rovnost, nerovnost znaků) se v tomto případě dále animoval i po kliknutí na „Finish“. Jinak vše fungovalo dobře a intuitivně.*“

Zoznam prevedených úprav:

- Nadpis grafu bol upravený, aby bolo jasné, akú štatistiku daný graf vyjadruje.

  > „*Přidal bych krátký popisek, aby se nový uživatel mohl rychleji zorientovat a zjistil, co v grafu vlastně může vidět.*“

- Zobrazenie informácie, že prebieha výpočet a nastavenie vyššej predvolenej rýchlosti.

  > „*Najprv som mala nastavení rýchlosť 1x a nechápala som, že nasleduje animácia, šlo to príliš pomaly a myslela som si, že sa nič nedeje. Možno by bolo lepšie nastaviť na vyššiu predvolenú rýchlosť, alebo nejakým spôsobom indikovať, že sa prehráva animácia.*“

- Zobrazenie rýchlosti vypĺňania tabuľky v časových jednotkách.

  „*Možno by bolo fajn vyjadriť rýchlosť v časových jednotkách miesto násobkov, neviem čoho násobky to sú.*“

Okrem spomenutých sa objavili aj 2 podnety, ktoré síce dávali zmysel, ale úpravy neboli prevedené z určitých dôvodov. Jedným bolo pridanie krokovania priamo v zdrojovom kóde algoritmu. Toto by vyžadovalo ďalšiu implementáciu, ktorá nie je triviálna, vyžiadala by si mnoho času a nebola uvedená v požiadavkách na výslednú aplikáciu. Na druhú stranu tento podnet môže poslúžiť ako inšpirácia do ďalšej fázy vývoja.

Druhým podnetom bolo, že pri vysokom počte rekurzívnych volaní sa môžu hodnoty pre ostatné vstupy, pri ktorých je počet volaní nízky, zliať do jednej čiary na ose X. S týmto problémom sa počítalo už pri návrhu aplikácie. Práve preto sú štatistiky zhrnuté aj v tabuľke, kde sú všetky hodnoty jednoducho čitateľné.

5.3 Experimenty prevedené na optimalizačných úlohách

Dôležitou súčasťou aplikácie sú štatistiky, ktoré majú poukazovať na výhody dynamického programovania. Preddefinované vstupné hodnoty, pre ktoré sú vypočítané štatistiky, neboli vybrané náhodne. Výberu predchádzali experimenty, ktorých cieľom bolo nájsť zaujímavé vstupné hodnoty, ktoré potvrdzujú (alebo vyvracajú) časovú zložitosť rekurzívneho a DP algoritmu.

5.3.1 Minimálny počet mincí

V rekurzívnom algoritme tejto úlohy závisí počet rekurzívnych volaní od konkrétnych hodnôt mincí. Je to z toho dôvodu, že sa v algoritme nachádza podmienka, ktorá určuje, či rekurzívne volanie nastane alebo nie. Podobne je to aj pri DP algoritme, kde sa podmienka nachádza vo vnútornom cykle. Na základe experimentov v tabuľke 5.1 sa pri rekurzívnom algoritme maximálny teoretický rovná reálnemu počtu volaní funkcie v prípade, kedy sa v zadaní nachádza minca, ktorá samotná má požadovanú hodnotu, a zároveň aj mince so všetkými menšími hodnotami. Aj keď je jasné, že jediná minca s hodnotou 8 stačí na vytvorenie hodnoty 8, algoritmus skúma aj ostatné mince, ktoré určite nebudú dávať optimálne riešenie (experiment č.8). Pri DP algoritme sa maximálna teoretická a reálna hodnota rovnajú len v prvých dvoch experimentoch, pretože sa tu nachádza iná podmienka ako v rekurzívnom algoritme.

V oboch algoritmoch na poradí mincí nezáleží, čo dokazujú experimenty č.8 a č.9. Ak sa na vstupe nachádzajú mince s hodnotou väčšou ako je požadovaná, počet volaní/behov cyklu sa nemení (experimenty č.10 a č.11). Rekurzívny algoritmus môže byť v niektorých prípadoch rýchlejší, čo dokazujú experimenty č.12 (malý počet mincí) a č. 13 (úloha nemá riešenie). V experimente č. 14 naopak vidno, aký je rekurzívny algoritmus neefektívny.

V tomto prípade navyše experimenty pomohli k zisteniu konkrétnej funkcie časovej zložitosti, pretože v literatúre k úlohe bolo uvedené len to, že trieda zložitosti je exponenciálna. Kvôli podobnosti s úlohou *Rezanie tyče* som predpokladal, že aj funkcia časovej zložitosti bude podobná, ak nie rovnaká. Preto bola pri experimentoch použitá rovnaká funkcia. Experimenty dokázali, že tento predpoklad bol správny. Časovú zložitosť skutočne vyjadruje funkcia $O(2^V)$.

	Zadanie		Rekurzívny algoritmus		DP algoritmus	
	Hodnota	Mince	teor. p.	reál. p.	teor. p.	reál. p.
1.	0	1	1	1	0	0
2.	1	1	2	2	1	1
3.	2	1,2	4	4	4	3
4.	3	1,2	8	7	6	5
5.	3	3	8	2	3	1
6.	4	1,2,3,4	16	16	16	10
7.	8	1,3,5	256	52	24	18
8.	8	1,2,3,4,5,6,7,8	256	256	64	36
9.	8	8,7,6,5,4,3,2,1	256	256	64	36
10.	10	1,2,5,10	1024	310	40	26
11.	10	1,2,5,10,11,12	1024	310	60	26
12.	12	4,6,8	4096	11	36	21
13.	10	7,4,8	1024	5	30	14
14.	30	1,2,3,4,10,15,25	1073741824	425132866	210	157

Tabuľka 5.1: Zadania a výsledky experimentov pre úlohu *Minimálny počet mincí*. Vysvetlivky: teor. - teoretický, reál. - reálny, p. - počet.

5.3.2 Rezanie tyče

Keďže v rekurzívnom algoritme nie je žiadna podmienka, počet volaní sa bude presne rovnať maximálnej teoretickej hodnote zo vzorca, čo dokazujú aj experimenty v tabuľke 5.2. Nezáleží ani na cenách jednotlivých dĺžok, iba na ich počte, čo dokazujú experimenty č.3 a č.4. DP algoritmus bude mať oproti teoretickej hodnote počet behov cyklu nižší, pretože ten závisí od cien jednotlivých dĺžok (v algoritme sa nachádza podmienka). Jediný prípad, kedy sa reálna hodnota rovná teoretickej je experiment č.1. Z experimentov vyplýva, že rekurzívny algoritmus nie je rýchlejší ako DP algoritmus ani pri veľmi jednoduchom vstupe.

	Zadanie	Rekurzívny algoritmus		DP algoritmus	
	Ceny	teor. p.	reál. p.	teor. p.	reál. p.
1.	1	2	2	1	1
2.	1,2,3,4	16	16	16	10
3.	2,3,4,4,6,6,11	128	128	49	29
4.	3,5,6,7,9,11,12	128	128	49	29
5.	1,2,3,4,5,6,7,8,9,10	1024	1024	100	55
6.	10,9,8,7,6,5,4,3,2,1	1024	1024	100	55
7.	1..20	1048576	1048576	400	210

Tabuľka 5.2: Zadania a výsledky experimentov pre úlohu *Rezanie tyče*. Vysvetlivky: teor. - teoretický, reál. - reálny, p. - počet.

5.3.3 Najdlhší spoločný podreťazec

Na základe experimentov v tabuľke 5.3 sa dá povedať, že funkcia časovej zložitosti u rekurzívneho algoritmu je naozaj len vrchný odhad. Tomu síce odporuje hneď prvý experiment, kde je maximálny teoretický počet nižší ako reálny. V tomto experimente však ide o extrémny prípad, kedy sú oba reťazce prázdne. Maximálny teoretický a reálny počet volaní sa rovná len v experimente č.2. Aj keď sú reťazce úplne rozdielne, skutočná hodnota sa teoretickej pri rekurzívnom algoritme veľmi nepribližuje (experiment č.5).

Vysoký počet volaní aj pri dvoch rovnakých reťazcoch je dôkazom, že rekurzívny algoritmus je veľmi neefektívny. DP algoritmus je rýchlejší vždy. V oboch algoritmoch nezáleží, v akom poradí boli reťazce zadané, čo dokazujú experimenty č.5 a č.6.

Teoretický aj reálny počet behov cyklu sa pri DP algoritme vždy rovná, pretože algoritmus vždy porovnáva každý znak reťazca X so všetkými znakmi druhého reťazca. K tomu je potrebné prejsť celú tabuľku.

	Zadanie		Rekurzívny algoritmus		DP algoritmus	
	Reťazec X	Reťazec Y	teor. p.	reál. p.	teor. p.	reál. p.
1.	”	”	$0.\overline{3}$	1	0	0
2.	’a’	”	1	1	0	0
3.	’aaaaa’	”	81	1	0	0
4.	”	’bbbbb’	81	1	0	0
5.	’aaaaa’	’bbbbb’	19683	503	25	25
6.	’bbbbb’	’aaaaa’	19683	503	25	25
7.	’aaaaa’	’aaaaa’	19683	2524	25	25
8.	’123here123’	’123here123’	1162261467	1621253	100	100
9.	’samesame1’	’samesame2’	129140163	285759	81	81
10.	’someString’	’anotherstring’	31381059609	6010094	130	130
11.	’ShortStr’	’LongestString’	3486784401	668540	104	104
12.	’test’	’twoTests’	177147	1735	32	32

Tabuľka 5.3: Zadania a výsledky experimentov pre úlohu *Najdlhší spoločný podreťazec*. Vysvetlivky: teor. - teoretický, reál. - reálny, p. - počet.

5.3.4 Editačná vzdialenosť

Podobne ako pri úlohe *Najdlhší spoločný podreťazec*, aj tu experimenty (tabuľka 5.4) ukázali, že funkcia zložitosti je vrchný odhad. Časová zložitosť sa značne zvyšuje s dĺžkou oboch reťazcov. Súčet dĺžok reťazcov v poslednom experimente je 20, pričom reálny počet volaní je 8-ciferné číslo oproti 121 opakovaní cyklu pri DP algoritme. Rekurzívny algoritmus je rýchlejší ako DP algoritmus napríklad v prípade veľmi krátkych reťazcov (experiment č.3), alebo ak je jeden z reťazcov prázdny (experiment č.4). Experiment č.8 ukazuje, že rekurzívny algoritmus je rýchlejší aj v prípade rovnakých reťazcov. V oboch algoritmoch nezáleží, v akom poradí boli reťazce zadané, čo dokazujú experimenty č.6 a č.7. Teoretický aj reálny počet behov cyklu sa pri DP algoritme vždy rovná. Rovnako ako pri úlohe *Najdlhší spoločný podreťazec* algoritmus prejde celou tabuľkou.

	Zadanie		Rekurzívny algoritmus		DP algoritmus	
	Reťazec X	Reťazec Y	teor. p.	reál. p.	teor. p.	reál. p.
1.	”	”	$0.\overline{3}$	1	1	1
2.	’a’	”	1	1	2	2
3.	’A’	’Ad’	9	5	6	6
4.	’aaaaa’	”	81	1	6	6
5.	”	’bbbbb’	81	1	16	16
6.	’aaaaa’	’bbbbb’	19683	2524	36	36
7.	’bbbbb’	’aaaaa’	19683	2524	36	36
8.	’aaaaa’	’aaaaa’	19683	6	36	36
9.	’123here123’	’123here123’	1162261467	11	121	121
10.	’samesame1’	’samesame2’	129140163	87372	100	100
11.	’someString’	’anotherstring’	31381059609	1172	154	154
12.	’gra’	’bagra’	2187	4	24	24
13.	’AMatchHere’	’tenisMatch’	1162261467	10195047	121	121

Tabuľka 5.4: Zadania a výsledky experimentov pre úlohu *Editačná vzdialenosť*. Vysvetlivky: teor. - teoretický, reál. - reálny, p. - počet.

5.3.5 Optimálny binárny vyhľadávací strom

Výsledky experimentov v tabuľke 5.5 sú špecifické z toho dôvodu, že pri tejto úlohe, na rozdiel od ostatných, bola uvedená časová zložitosť pomocou Theta notácie. To znamená, že by mala vyjadrovať presný počet volaní funkcie.

Teoretický a reálny počet volaní sa najviac približuje v experimente č.2. V experimentoch č.3 až č.6 je reálny počet vyšší od teoretického. V experimente č.7 sa počty opäť približujú. V experimentoch č.8 a č.9 už je teoretický počet výrazne vyšší od reálneho.

Aj pri malom počte kľúčov je DP algoritmus rýchlejší ako rekurzívny. Pri oboch algoritmoch nezáleží na počtoch vyhľadaní, iba na počte kľúčov. V algoritmoch sa totiž nenachádza podmienka pre volanie/vykonanie cyklu, ktorá by brala do úvahy počet vyhľadaní.

Reálny počet behov cyklu v DP algoritme sa líši od maximálnej teoretickej hodnoty. Ide teda o vrchný odhad. Aby sa teoretický a reálny počet rovnali, musel by byť každý cyklus vykonaný presne N-krát, čo však v algoritme neplatí.

	Zadanie	Rekurzívny algoritmus		DP algoritmus	
	Počty vyhľadaní kľúčov	teor. p.	reál. p.	teor. p.	reál. p.
1.	1	4	1	1	0
2.	1,4	6	5	8	2
3.	1,4,7	12	15	27	7
4.	1,4,7,9	32	45	64	16
5.	1,2,3,4,5,6,7,8	2896	3645	512	112
6.	1,2,3,4,55,66,77,88	2896	3645	512	112
7.	1,2,3,4,5,6,7,8,9,10	33159	32805	1000	210
8.	1,2,3,4,5,6,7,8,9,10,11	114966	98415	1331	275
9.	1,2,3,4,5,6,7,8,9,10,11,12,13,14,15	18482596	7971615	3375	665

Tabuľka 5.5: Zadania a výsledky experimentov pre úlohu *Optimálny binárny vyhľadávací strom*. Vysvetlivky: teor. - teoretický, reál. - reálny, p. - počet.

Kapitola 6

Záver

Cieľom tejto práce bolo vytvoriť webovú aplikáciu na podporu výuky techniky návrhu algoritmov, ktorá sa nazýva dynamické programovanie. Tento cieľ bol splnený, pričom výsledná aplikácia je voľne dostupná na adrese https://dp-learn.firebaseapp.com/.

V prvom rade bolo treba túto techniku dôkladne preštudovať a vybrať konkrétne úlohy, ktoré sa pomocou nej dajú riešiť. Následne prebehla analýza existujúcich riešení. Dôvodom bolo, aby aplikácia nielenže spĺňala základné požiadavky zo zadania práce, ale aj doplnila funkcionality, ktoré chýbajú v doterajších riešeniach. Každá z webových stránok a aplikácií, ktoré sa venujú dynamickému programovaniu, ponúkajú len konkrétnu funkcionalitu (napríklad teóriu alebo naopak iba grafické znázornenie behu algoritmu). Aplikácia DP learn bude poskytovať všetku funkcionalitu na jednom mieste.

Aby aplikácia vyzerala moderne a bola zároveň rýchla a spoľahlivá, bolo treba vybrať vhodné vývojové technológie. Výsledkom je komplexná aplikácia, postavená na knižnici pre vývoj užívateľských rozhraní s názvom React. Implementovaná bola v jazyku TypeScript.

Kľúčovou časťou aplikácie je grafické znázornenie priebehu algoritmu, ktorý bol navrhnutý technikou dynamického programovania pri využití prístupu zdola-hore. Okrem toho aplikácia poskytuje dôležitú teóriu k dynamickému programovaniu a aj konkrétnym optimalizačným úlohám. Nechýbajú ani zdrojové kódy algoritmov v jazyku C. Aplikácia navyše dokáže vykresľovať grafy, ktoré porovnávajú efektivitu rekurzívneho a DP algoritmu. Túto funkcionalitu neposkytuje žiadne z existujúcich riešení.

Okrem toho, že bola aplikácia priebežne testovaná v priebehu vývoja, prebehlo aj užívateľské testovanie formou dotazníka. Toto testovanie bolo veľmi prospešné. Na základe výsledkov bolo prevedených niekoľko zmien, vďaka ktorým dostala aplikáciu svoju výslednú podobu.

Po dokončení tejto práce by som sa chcel sústrediť na spropagovanie aplikácie. Myslím, že má potenciál poslúžiť komukoľvek, kto sa zaujíma o dynamické programovanie. K tomu je treba v prvom rade zvýšiť výkonnosť vyhľadávania aplikácie vo webových vyhľadávačoch. Ďalším krokom by mohlo byť napríklad spomenúť aplikáciu na webových portáloch, ktoré sa zaoberajú programovacími technikami.

Aplikácia v súčasnej podobe obsahuje 5 optimalizačných úloh. Ďalších úloh, ktoré sa dajú riešiť dynamickým programovaním je veľký počet. V budúcnosti by mohla byť aplikácia rozšírená o ďalšie takéto úlohy.

Pri dynamickom programovaní sa okrem prístupu zdola-hore, na ktorú sa aplikácia sústreďuje, dá navrhnúť algoritmus aj prístupom zhora-dole. Do časti aplikácie s porovnaním algoritmov by bolo vhodné pridať aj tento typ algoritmu, prípadne aj jeho zdrojový kód do časti s teóriou.

Pri užívateľskom testovaní sa objavil návrh na pridanie možnosti krokovania priamo v zdrojovom kóde algoritmu. Ide o náročnú požiadavku, ktorú nebolo možné splniť z časových dôvodov. To však nevylučuje jej pridanie v ďalšej fáze vývoja, práve naopak by mohlo ísť o kľúčovú požiadavku.

Literatúra

[1] *What is Angular?* Angular, [Online; navštívené 13.05.2019].
URL https://angular.io/docs

[2] Bellman, R. E.: *Eye of the Hurricane: An Autobiography*. World Scientific Pub Co Inc, 1984, ISBN 978-9971966003.

[3] Boneh, D.: *Advances in Cryptology – CRYPTO 2003*. Springer, 2003, ISBN 978-3-540-45146-4.

[4] Chinda, G.: *A complete guide to default props in React*. Log Rocket, [Online; navštívené 05.05.2019].
URL https://blog.logrocket.com/a-complete-guide-to-default-props-in-react-984ea8e6972d

[5] *React Lifecycle Methods: render and componentDidMount*. Codin Game, [Online; navštívené 05.05.2019].
URL https://www.codingame.com/playgrounds/8747/react-lifecycle-methods-render-and-componentdidmount

[6] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; aj.: *Introduction to Algorithms*. The MIT Press, 3 vydanie, 2009, ISBN 978-0-262-03384-8.

[7] *5 Reasons Why React.js Is So Popular*. Cygnis Media, [Online; navštívené 05.05.2019].
URL https://www.cygnismedia.com/blog/reasons-why-react-js-is-popular/

[8] *Firebase Hosting*. Firebase, [Online; navštívené 06.05.2019].
URL https://firebase.google.com/docs/hosting

[9] Galles, D.: *Dynamic Programming (Making Change)*. University of San Fransisco, [Online; navštívené 15.05.2019].
URL https://www.cs.usfca.edu/~galles/visualization/DPChange.html

[10] *Find minimum number of coins that make a given value*. GeeksforGeeks, [Online; navštívené 26.04.2019].
URL https://www.geeksforgeeks.org/find-minimum-number-of-coins-that-make-a-change/

[11] *Longest Common Substring | DP-29*. GeeksforGeeks, [Online; navštívené 26.04.2019].
URL https://www.geeksforgeeks.org/longest-common-substring-dp-29/

[12] *Optimal Binary Search Tree | DP-24*. GeeksforGeeks, [Online; navštívené 27.04.2019].
URL https://www.geeksforgeeks.org/optimal-binary-search-tree-dp-24/

[13] *Suffix Tree Application 5 – Longest Common Substring*. GeeksforGeeks, [Online; navštívené 26.04.2019].
URL https://www.geeksforgeeks.org/suffix-tree-application-5-longest-common-substring-2

[14] *EditDistance.java*. GitHub, [Online; navštívené 13.05.2019].
URL https://github.com/mission-peace/interview/blob/master/src/com/interview/dynamic/EditDistance.java

[15] *Migrating Multi-page Web Applications to Single-page AJAX Interfaces*, IEEE, Apríl 2007, ISBN 0-7695-2802-3.

[16] *PHP2Uni: Building Unikernels Using Scripting Language Transpilation*, IEEE, Apríl 2017, ISBN 978-1-5090-5817-4.

[17] *Features*. CSSinJSS, [Online; navštívené 15.05.2019].
URL https://cssinjs.org/features?v=v10.0.0-alpha.16

[18] Karásek, J.: *Material Design: Nový designový jazyk Androidu (L Preview), který má sjednotit vzhled aplikací*. smartmania, [Online; navštívené 05.05.2019].
URL https://smartmania.cz/material-design-novy-designovy-jazyk-androidu-l-preview-ktery-ma-sjednotit-vzhled-aplikaci-8115/

[19] *Big-θ (Big-Theta) notation*. Khan Academy, [Online; navštívené 11.05.2019].
URL https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-big-theta-notation

[20] Knuth, D. E.: BIG OMICRON AND BIG OMEGA AND BIG THETA. *SIGACT News*, May 1976: str. 7.

[21] Kuo, W.; Zuo, M. J.: *Optimal Reliability Modeling: Principles and Applications*. Wiley, 2002, ISBN 978-0471397618.

[22] Maison, A.: *Dynamic Programming or 'Divide and Conquer'*. DZone, [Online; navštívené 27.04.2019].
URL https://dzone.com/articles/dynamic-programming-or-quotdivide-and-conquerquot/

[23] Mareš, M.; Valla, T.: *Průvodce labyrintem algoritmů*. CZ.NIC, 2017, ISBN 978-80-88168-22-5.

[24] Meyer, E. A.: *CSS Pocket Reference: Visual Presentation for the Web*. O'Reilly Media, 5 vydanie, 2018, ISBN 978-1492033394.

[25] Moore, K.; Chumbley, A.: *Complexity Classes*. BRILLIANT, [Online; navštívené 25.04.2019].
URL https://brilliant.org/wiki/complexity-classes/

[26] Ravichandran, A.: *Props and State in React Native explained in Simple English*. codeburst, [Online; navštívené 05.05.2019].
URL https://codeburst.io/props-and-state-in-react-native-explained-in-simple-english-8ea73b1d224e

[27] *Conditional Rendering*. React, [Online; navštívené 08.05.2019].
URL https://reactjs.org/docs/conditional-rendering.html

[28] *React.Component*. React, [Online; navštívené 08.05.2019].
URL
https://reactjs.org/docs/react-component.html?utm_source=caibaojian.com

[29] Salian, V.: *Recursion – How to overflow the stack and how not to*. Medium, [Online; navštívené 26.04.2019].
URL https://medium.com/@vijeshsalian/recursion-how-to-overflow-the-stack-and-how-not-to-b9dcffdfab27

[30] *Complexity of edit distance (Levenshtein distance) recursion top down implementation*. Stack Overflow, [Online; navštívené 10.05.2019].
URL https://stackoverflow.com/questions/14616339/complexity-of-edit-distance-levenshtein-distance-recursion-top-down-implementa

[31] *Difference between Top-down and Bottom-up Approach*. TechDifferences, [Online; navštívené 26.04.2019].
URL https://techdifferences.com/difference-between-top-down-and-bottom-up-approach.html

[32] *Dynamic Programming – Edit Distance Problem*. Tutorial Horizon, [Online; navštívené 27.04.2019].
URL https://algorithms.tutorialhorizon.com/dynamic-programming-edit-distance-problem/

[33] *Rod-Cutting*. Vamonos, [Online; navštívené 15.05.2019].
URL http://rosulek.github.io/vamonos/demos/rod_cutting.html

[34] *What is Vue.js?* Vue.js, [Online; navštívené 13.05.2019].
URL https://vuejs.org/v2/guide/

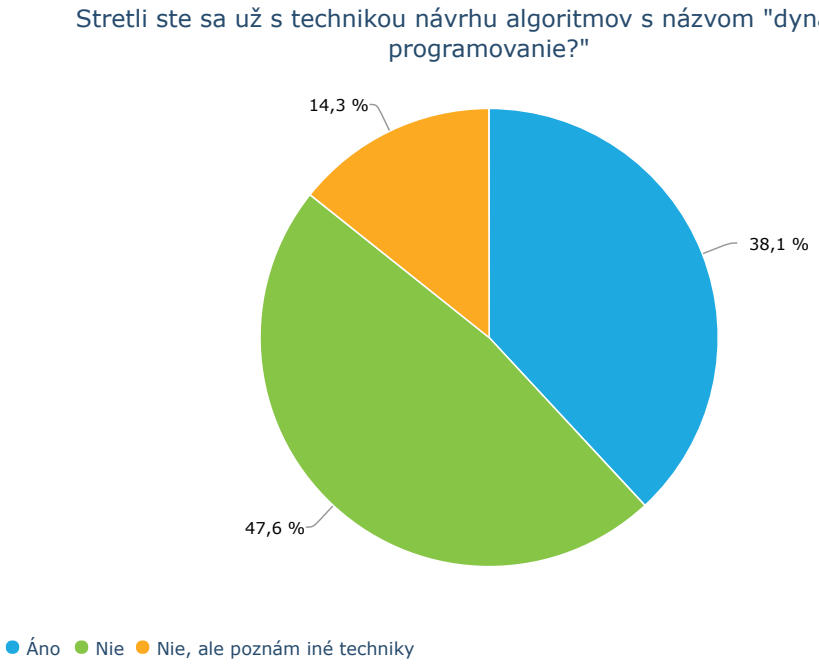
[35] Witmer, D.: *Lecture 7: Dynamic Programming I: Optimal BSTs*. Carnegie Mellon University, [Online; navštívené 10.05.2019].
URL http:
//www.cs.cmu.edu/afs/cs/academic/class/15750-s17/ScribeNotes/lecture7.pdf

Príloha A

Výsledky užívateľského testovania

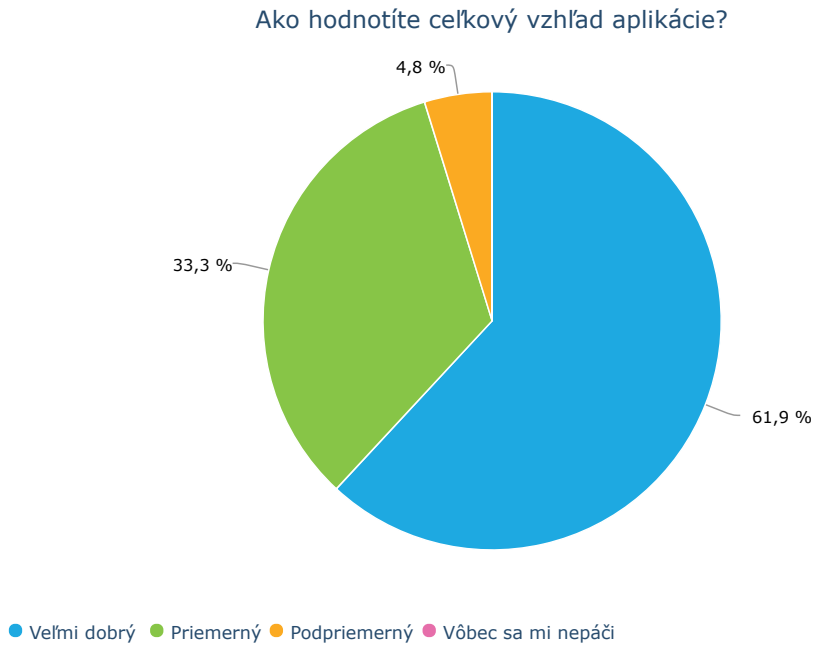
V tejto prílohe sa nachádza výber niektorých zaujímavých grafov s výsledkami užívateľského testovania.

A.1 Počet užívateľov, ktorí sa stretli s dynamickým programovaním



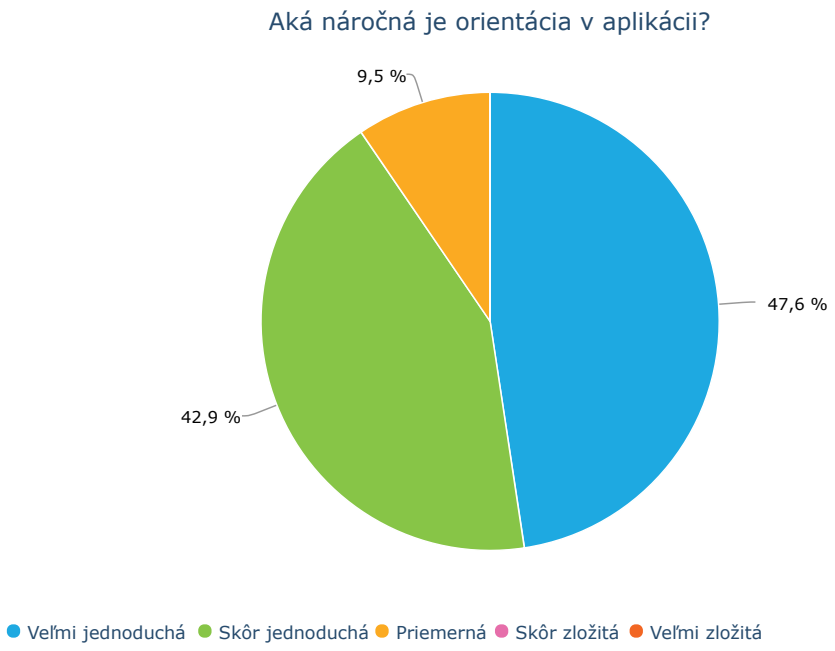
Obr. A.1:

A.2 Hodnotenie celkového vzhľadu aplikácie



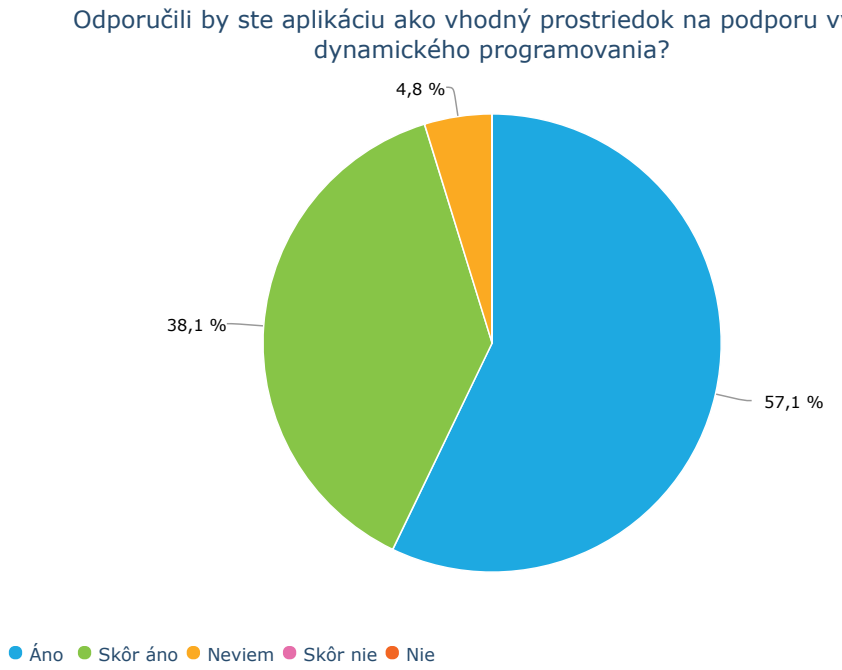
Obr. A.2:

A.3 Hodnotenie náročnosti orientácie v aplikácii



Obr. A.3:

A.4 Odpovede na otázku, či by užívateľ odporučil aplikáciu



Obr. A.4:

Príloha B

Obsah CD

- Súbor README.txt s popisom obsahu jednotlivých adresárov CD
- Návod na inštaláciu aplikácie
- Zdrojový kód aplikácie
- Závislosti potrebné pre preklad
- Preložená aplikácia
- Zdrojový kód technickej správy
- PDF s technickou správou