Увод у динамичко програмирање

***Андреја Илић*, Ниш**

Овај и наредни број Тангенте садржаће кратак увод у Динамичко програмирање. Поред уводне приче о идеји, биће приказани неки од познатијих проблема из ове класе који на леп начин покривају основне методе. Сва питања и коментаре можете слати на адресу аутора[1]. Уживајте.

1. Увод

Један од честих алгоритамских проблема јесте проблем оптимизације: задати проблем може имати више решења, свако решење има своју вредност, а тражи се оно које има екстремну вредност. У једној широкој класи проблема оптимизације, решење можемо наћи коришћењем **динамичког програмирања** (енг. dynamic programming).

Теоријски гледано проблеме оптимизације, као и проблеме претраге, можемо решити симулацијом свих стања, односно обиласком целокупног простора претраге (енг. backtracking). Ово је наравно коректан приступ, међутим он није употребљив за проблеме са великим бројем стања. Такође, можемо користити **грабљиви приступ** (енг. greedy algorithm), који се заснива на бирању локалног најбољег потеза у свакој тачки гранања. Ова врста хеуристике може знатно смањити сложеност и простор претраге, али се не може тврдити њена тачност.

Идеја динамичког програмирања је да се искористи принцип домина: све нанизане домине ће попадати ако се поруши прва домина у низу. Дакле, да би се решио неки проблем, треба решити неки његов мањи случај (потпроблем), а затим показати како се решење задатог проблема може конструисати полазећи од (решених) потпроблема. Овакав приступ је базиран на математичкој индукцији.

2. Шта је динамичко програмирање?

Динамичко програмирање је назив популарне технике у програмирању којом драстично можемо смањити сложеност алгоритма: од експоненционалне до полиномијалне. Реч „програмирање“ у самом називу технике се односи на тзв. таблични метод, а не на само куцање компјутерског кода. Слично методи **„подели па владај“** (енг. divide and conquer), динамичко програмирање решавање једног проблема своди на решавање потпроблема. За овакве проблеме се каже да имају **оптималну структуру** (енг. optimal substructure). „Подели па владај“ алгоритми врше партицију главног проблема на независне потпроблеме. Затим наступа рекурзивно решавање потпроблема, како би се њиховим спајањем добило решење полазног проблема. Алгоритам који је добар представник ове класе јесте **сортирање учешљавањем** (енг. merge sort), алгоритам за сортирање низа.

Динамичко програмирање (ДП) такође врши партицију главног проблема на потпроблеме (енг. overlapping subproblems), али који нису независни. Још једна битна

[1] andrejko.ilic@gmail.com

карактеристика ДП јесте да се сваки потпроблем решава највише једном, чиме се избегава поновно рачунање нумеричких карактеристика истог стања (одатле „таблични метод“, из простог разлога што се, како ћемо убрзо видети, решења потпроблема чувају у помоћним таблицама).

Описану меморизацију објаснимо на примеру Фибоначијевих бројева ($F_1 = F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$ за $n > 2$). Наивна имплементација рекурзивне функције за рачунање n-тог Фибоначијевог броја базирана на дефиницији је приказана у алгоритму А.

Алгоритам А. Функција $Fib(n)$ за рачунање n-тог Фибоначијевог броја

```
Input: prirodni broj n
Output: n-ti Fibonačijev broj
1 if n ≤ 2 then
2 |   return 1;
3 endif
4 return (Fib(n − 1) + Fib(n − 2));
```

Приметимо да при рачунању вредности $Fib(n)$, функције $Fib(m)$ за $m < n$ се позивају већи број пута. За $n = 5$ имали бисмо следеће:

$$\begin{aligned} Fib(5) &= Fib(4) + Fib(3) \\ &= (Fib(3) + Fib(2)) + (Fib(2) + Fib(1)) \\ &= ((Fib(2) + Fib(1)) + Fib(2)) + (Fib(2) + Fib(1)). \end{aligned}$$

Међутим уколико бисмо вршили меморизацију израчунатих стања, не бисмо морали да „улазимо“ у рекурзије бројева које смо већ рачунали. Дефинишимо низ $fib[k]$ у коме ћемо памтити Фибоначијеве бројеве које рачунамо у рекурзивним позивима. На почетку иницијализујмо све елементе низа на -1 (чиме их практично маркирамо као неизрачунате). Овим приступом избегавамо непотребна рачунања.

Алгоритам Б. Функција $Fib(n)$ за рачунање n-тог Фибоначијевог броја са меморизацијом

```
Input: prirodni broj n
Output: n-ti Fibonačijev broj
1 if fib[n] ≠ −1 then
2 |   return fib[n];
3 endif
4 if n ≤ 2 then
5 |   fib[n] = 1;
6 endif
7 fib[n] = Fib(n − 1) + Fib(n − 2);
8 return fib[n];
```

Грубо говорећи, структуру алгоритма базираном на ДП, можемо описати у четири корака:

- окарактерисати структуру оптималног решења;
- рекурзивно дефинисати оптималну вредност;
- израчунати оптималну вредност проблема „одозго на горе";
- реконструисати оптимално решење.

Као резултат прва три корака добијамо оптималну вредност. Последњи корак извршавамо уколико је потребно и само оптимално решење. При његовој реконструкцији обично користимо неке додатне информације које смо рачунали у кораку 3.

Вероватно смо сви још у основној школи наишли на стандардни проблем кусура: продавац има дискретан скуп новчаница са вредностима $1 = v_1 < v_2 < \cdots < v_n$ (бесконачно од сваке врсте) и треба вратити кусур у вредности од S новчаних јединица, при чему треба минимизирати број новчаница. Уколико је скуп новчаница дискретан и мења се само вредност кусура S, проблем можемо решити рекурзијом односно преко n угњеждених петљи. Међутим, шта се дешава у случају када ни вредности новчаница, као ни њихов број нису познати унапред. Овај и многе друге проблеме можемо једноставно решити уз помоћ ДП-а, док је овај проблем једна верзија проблема ранца који ћемо обрадити у поглављу 4.

Како бисмо боље разумели концепт ДП проћи ћемо детаљно кроз наредни проблем.

Проблем 1. [***Максимална сума несуседних у низу***] *Дат је низ a природних бројева дужине n. Одредити подниз датог низа чији је збир елемената максималан, а у коме нема суседних елемената.*

Потпроблем горњег проблема можемо дефинисати као: налажење траженог подниза на неком делу полазног низа a. Прецизније, дефинисаћемо нови низ d на следећи начин:

$$d[k] = \text{максималан збир несуседних елемената низа } (a_1, a_2, \ldots, a_k) \text{ за } k \in [1, \ldots, n].$$

Решење полазног проблема ће бити вредност $d[n]$.

Напомена. Алгоритам за дефинисање потпроблема не постоји и зависи од саме природе проблема. Некада морате проћи неколико могућности како бисте дошли до праве. Наравно, вежбом и детаљнијом анализом проблема може се стећи нека „интуиција", али универзално решење не постоји („There is no free lunch").

Дефинишимо сада вредности низа d рекурзивно. Претпоставимо да смо израчунали вредности низа d до $(k-1)$-ог елемента и желимо израчунати k-ти. После додавања новог елемента a_k, за тражену оптималну вредност подниза $(a_1, a_2, \ldots, a_k)$ имамо две могућности: елемент a_k укључујемо у тражену суму или не (елементе са индексима $1, \ldots, k-2$ користимо у оба случаја, јер на њих не утиче чињеница да ли је елемент k ушао у подниз или не). Уколико га укључимо тада елемент на месту $k-1$ не сме ући у суму (пошто је суседан са k-тим); у супротном елемент a_{k-1} можемо укључити. Ово формалније записујемо као:

$$d[k] = \max\{d[k-1], a_k + d[k-2]\} \text{ за } k \geqslant 3 \tag{1}$$

Дефинисањем базе $d[1] = \max\{0, a_1\}$ и $d[2] = \max\{0, a_1, a_2\}$ вредности низа d можемо израчунати једноставним for циклусом. При рачунању вредности $d[n]$ преко формуле (1), рекурзивним позивима бисмо могли ићи у бесконачност уколико неке од вредности низа d немамо израчунате без рекурзије. У нашем случају, елементи са индексима 1 и 2 се могу иницијализовати на горе описан начин. Два узастопна елемента низа d нам дефинишу базу,

зато што вредност елемента са индексом n зависи само од претходна два елемента. На пример, у случају да је $d[k] = \max\{d[k-1], d[k-3]\} + 4$ базу морају да чине прва три елемента низа d.

Напомена. Дефинисање низа d је логичка последица горње индуктивне анализе. Испитујући случајеве последњег елемента низа добили смо једноставан услов оптималне вредности.

Алгоритам Ц. Псеудо код проблема максималне суме несуседних у низу

```
    Input: niz a prirodnih brojeva dužine n
    Output: subsequence - podniz nesusednih elemenata sa najvećom sumom
 1  if n = 1 then
 2  |   return subsequence = a;
 3  endif
 4  d[1] = max{0, a_1};
 5  d[1] = max{0, a_1, a_2};
 6  for k ← 3 то n do
 7  |   if d[k − 1] > d[k − 2] + a[k] then
 8  |   |   d[k] = d[k − 1];
 9  |   else
10  |   |   d[k] = d[k − 2] + a[k];
11  |
12  endfor
13  subsequence = ∅;
14  currentIndex = n;
15  while (currentIndex > 0) do
16  |   if d[currentIndex] ≠ d[currentIndex − 1] then
17  |   |   add a[currentIndex] to subsequence;
18  |   |   currentIndex = currentIndex − 2;
19  |   else
20  |   |   currentIndex = currentIndex − 1;
21  |   endif
22  endw
23  return subsequence;
```

Описаним алгоритмом можемо израчунати максималну суму подниза несуседних елемената, али ћемо се задржати на реконструкцији самог подниза. У овом конкретном случају не морамо памтити додатне информације, пошто је рекурентна формула једноставна. За његову реконструкцију нам је потребно да знамо да ли је за вредност $d[k]$ елемент на позицији k ушао у подниз или не. Уколико би елемент $a[n]$ припадао траженом подనизу, тада би вредност $d[n]$ била једнака $d[n-2] + a[n]$. У случају да елемент са индексом n припада, тада испитујемо елемент са индексом $n-2$; у супротном елемент n не припада траженом подниzu, па зато прелазимо на елемент $a[n-1]$. Треба напоменути да описаним поступком добијамо оптимални подниз у обрнутом поретку.

Сложеност описаног алгоритма је линеарна, $O(n)$ (сложеност рачунања низа d је линеарна, док је реконструкција траженог подниза сложености $O(|subsequence|) = O(n)$). Уклапањем свих корака, алгоритам можемо описати псеудокодом у Алгоритму Ц.

На пример, за дати низ $a = \{1, -2, 0, 8, 10, 3, -11\}$ је $d = \{1, 1, 1, 9, 11, 12, 12\}$.

1	−2	0	8	10	3	−11

Слика 1: тражени подниз у примеру Проблема 1

Напомена. Описани алгоритам се може имплементирати и рекурзивно. Параметар рекурзивне функције је индекс k и она као резултат враћа вредност $d[k]$. Како бисмо избегли поновно рачунање истих стања, маркираћемо обиђена стања и у случају да је стање већ рачунато, вратићемо одмах вредност $d[k]$. Општи опис алгоритма дат је псеудо кодом у Алгоритму Д.

Алгоритам Д. Општи опис рекурзивне варијанте ДП-а

```
Input: stanje A
Output: optimalna vrednost stanja A, zapamćena u globalnom nizu d
1 if stanje A već obidjeno then
2 |   return d[A];
3 endif
4 inicijalizuj vrednost d[A] rekurzivno;
5 markiraj stanje A kao obidjeno;
6 return d[A];
```

3. Познатији проблеми

У овом одељку ћемо продискутовати неке познатије представнике класе проблема које можемо решити уз помоћ ДП. У примеру из претходног дела смо видели како можемо ДП применити на једно-димензионални проблем, где нам је потпроблем био окарактерисан једним параметром.

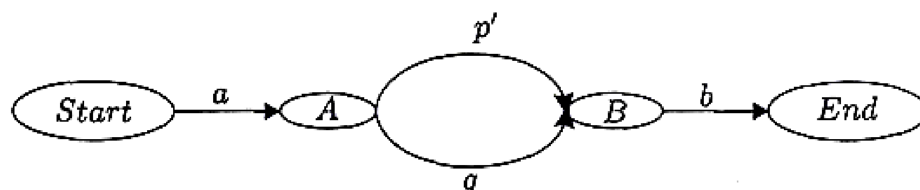
Проблем 2. [***Проблем максималног збира у матрици***] *Дата је матрица a димензије $n \times m$ попуњена целим бројевима. Са сваког поља у матрици дозвољено је прећи само на поље испод или на поље десно од тог поља (уколико постоје). Потребно је изабрати пут од горњег левог поља (поља са координатама $(1, 1)$), до доњег десног поља (поља са координатама (n, m)), тако да збир бројева у пољима преко којих се иде, буде максималан.*

Као што се може претпоставити, генерисање свих путева и памћење оног пута који има највећи збир, није добра идеја. Број могућих путева расте експоненцијално са порастом димензија матрице (прецизније, број различитих путева је једнак $\binom{n+m}{n}$).

Покушајмо да размишљамо уназад: нека је $P = \{p_1 = (1, 1), \ldots, p_{n+m-1} = (n, m)\}$ тражени пут са максималним збиром. Како се сваким потезом растојање до крајњег поља смањује за 1, знамо да је дужина траженог пута једнака $n + m - 1$. Тада сваки део оптималног пута P спаја полазно и завршно поље тог дела пута на оптималан начин. Ову чињеницу

можемо једноставно доказати контрадикцијом: претпоставимо да постоји оптималнији пут који спаја поља p_a и p_b од пута $p' = p_a, p_{a+1}, \ldots, p_b$. Означимо тај пут са q. Тада добијамо да је пут $p_1, p_2, \ldots, p_{a-1}, q, p_{b+1}, \ldots, p_{n+m-1}$ оптималнији од пута P, што је немогуће. На основу ове чињенице се просто намеће дефиниција потпроблема:

$$path[i][j] = \text{оптималан пут од поља } (1,1) \text{ до поља } (i,j),$$
$$d[i][j] = \text{сума елемената на оптималном путу од поља } (1,1) \text{ до поља } (i,j)$$



Слика 2: уколико је пут $a - p' - b$ оптималан за стања $Start$ и End, тада је и пут p' оптималан за стања A и B

Из горњих дефиниција имамо да ће крајњи резултат бити $d[n][m]$, односно $path[n][m]$. Питање које се сада поставља јесте да ли се $d[i][j]$ може рачунати рекурзивно? До поља (i, j) можемо доћи преко поља $(i-1, j)$ и $(i, j-1)$ (претпоставимо да поља постоје). Како је сваки део оптималног пута оптималан, $d[i][j]$ дефинишемо као:

$$d[i][j] = \max\{d[i-1][j], d[i][j-1]\} + a[i][j] \text{ за } i, j \geqslant 2$$

а тада се пут $path[i][j]$ добија додавањем поља (i, j) на онај од путева $path[i-1][j]$ или $path[i][j-1]$ који даје већи збир у горњој једначини.

Базу ДП ће у овом случају представљати елементи прве врсте и прве колоне матрице d пошто за њих не важи горња једначина. Како од поља $(1, 1)$ до поља $(1, j)$, односно $(i, 1)$, постоје јединствени путеви, попуњавање матрице се може једноставно реализовати на следећи начин:

$$d[1][1] = a[1][1],$$
$$d[1][j] = d[1][j-1] + a[1][j] \text{ за } j \geqslant 2,$$
$$d[i][1] = d[i-1][1] + a[i][1] \text{ за } i \geqslant 2,$$

што је јако слично рекурентној једначини, с тим што овде немамо бирање максимума из двоелементног скупа - јер елементи имају само једног суседа.

Као и у првом примеру, при реконструкцији самог пута, није потребно памтити целокупну матрицу $path$. Тражени пут се може пронаћи кретањем од назад и бирањем оног поља (горњег или левог) које нам доноси већи збир. Уколико меморијско ограничење то дозвољава, можемо да памтимо претходно поље са којег смо дошли, чиме бисмо избегли специјалне случајеве које нам скривају прва врста и колона.

Пут ће опет бити у обрнутом редоследу - па се или мора ротирати на крају или памтити у стеку. Међутим, како у овом случају знамо дужину пута (у првом проблему нисмо знали колика је дужина траженог подниза) можемо одмах и сам низ попуњавати од назад, па нећемо имати потребе за његовим ротирањем.

Напомена. Проблем са специјалним случајевима прве врсте и колоне можемо решити елегантније. Наиме, проблем рекурентне формуле је тај што наведена поља не морају увек постојати. Уколико бисмо додали још по једну врсту и колону са индексом 0 у матрици d и попунимо их бројевима који су сигурно мањи од бројева из матрице, тиме обезбеђујемо да пут форсирано остане у првобитној матрици. У овом случају базу ДП не бисмо имали, тј. редови $3, 4, \ldots, 10$ псеудо кода би били замењени иницијализацијом нулте врсте и колоне бројевима који су мањи од свих бројева у матрици (која су обично дата као ограничења самог проблема).

Алгоритам Е. Псеудо код проблема максималног збира у матрици

```
Input: matrica a celih brojeva dimenzije n × m
Output: path - niz elemenata matrice koji predstavlja traženi put
1  d[1][1] = a[1][1];
2  prior[1][1] = (0, 0);
3  for j ← 2 то m do
4  |   d[1][j] = d[1][j − 1] + a[1][j];
5  |   prior[1][j] = (1, j − 1);
6  endfor
7  for i ← 2 то n do
8  |   d[i][1] = d[i − 1][1] + a[i][1];
9  |   prior[i][1] = (i − 1, 1);
10 endfor
11 for i ← 2 то n do
12 |   for j ← 2 то m do
13 |   |   if d[i − 1][j] > d[i][j − 1] then
14 |   |   |   d[i][j] = d[i − 1][j] + a[i][j];
15 |   |   |   prior[i][j] = (i − 1, j);
16 |   |   else
17 |   |   |   d[i][j] = d[i][j − 1] + a[i][j];
18 |   |   |   prior[i][j] = (i, j − 1);
19 |   |   endif
20 |   endfor
21 endfor
22 path = ∅;
23 currentField = (n, m);
24 while (currentField ≠ (0, 0)) do
25 |   add currentField in path;
26 |   currentField = prior[currentField.i][currentField.j];
27 endw
28 return path;
```

На Слици 3 је приказан резултат алгоритама на једном примеру.

1	3	1	0	0
−11	4	10	−10	8
4	2	5	7	0
10	1	−2	1	1

матрица a

1	4	5	5	5
−10	8	18	−5	13
−6	10	23	30	30
4	14	21	31	32

матрица d решење потпроблема

1	3	1	0	0
−11	4	10	−10	8
4	2	5	7	0
10	1	−2	1	1

тражени пут

Слика 3: пример проблема максиманог збира у матрици

Овај проблем се може срести у више варијанти.

- Прва варијанта се разликује у додатном услову: тражени пут мора проћи кроз поље (x, y). Ово је један од честих услова ДП проблема, када (уз услов оптималности) решење мора проћи кроз нека стања. Решење оваквог проблема се обично своди на почетни проблем који независно позивамо над-делом улаза. Конкретно у нашем случају, тражени пут ће бити унија оптималних путева p и q, где је p пут од поља $(1, 1)$ до поља (x, y), а пут q од поља (x, y) до поља (n, m).

- Друга варијанта дозвољава да се из једног поља пређе не само у поља која су десно и доле, већ и у поље изнад. Наравно, овде се мора додати и услов да се свако поље у путу обиђе највише једном (у супротном би могло доћи до бесконачног пута). Овај проблем је компликованији од претходних и нећемо га овде разматрати.

Пре него што пређемо на наредни проблем, дефинишимо појам подниза. *Низ a је подниз низа b уколико се низ a може добити избацивањем неких елемената низа b.* На пример, имамо да је $\{2, 1, 7, 7\}$ подниза низа $\{5, 2, 1, 3, 7, 1, 7\}$, али није подниз од $\{2, 5, 7, 7, 1\}$.

Проблем 3. [***Најдужи заједнички подниз (НЗП)***] *Дата су два низа a и b. Наћи низ највеће могуће дужине који је подниз и за a и за b.*

Проблем најдужег заједничког подниза (енг. Longest Common Subsequence) јако добро демонстрира моћ ДП. На почетку, веома брзо можемо уочити да се укључивањем било каквих карактеристика проблема не може избећи експоненцијална сложеност, уколико проблем не посматрамо кроз потпроблеме. Најлакше можемо баратати поднизовима који су састављени од почетних елемената низа, па зато матрицу lcs дефинишемо као:

$$lcs[x][y] = \text{НЗП низова } \{a_1, \ldots, a_x\} \text{ и } \{b_1, \ldots, b_y\} \text{ за } 1 \leqslant x \leqslant n,\ 1 \leqslant y \leqslant m,$$

где су n и m дужине низова a односно b. У наставку ћемо подниз $\{a_1, \ldots, a_x\}$ низа a означавати са a^x. Покушајмо сада да изразимо везу међу елементима описане матрице. На почетку, НЗП за поднизове a^1 и b^k је $\{a_1\}$ уколико b^k садржи елемент $a^1 \equiv a_1$; иначе је $\varnothing$. Наравно, за све елементе матрице важи $length(lcs[x][y]) \leq \min\{x, y\}$.

Посматрајмо сада поднизове a^x и b^y. За њихов НЗП постоје две могућности: елемент $a[x]$ улази у НЗП или не. Уколико не улази, тада једноставно имамо да је $lcs[x][y] = lcs[x-1][y]$. Претпоставимо сада да $a[x]$ улази у НЗП – тада подниз b^y мора садржати елемент $a[x]$. Нека

се елемент $a[x]$ налази на позицијама $1 \leqslant c_1 < \cdots < c_k \leqslant y$ у поднизу b^y. Како елемент $a[x]$ мора имати свог парњака у низу b^y, имамо да је

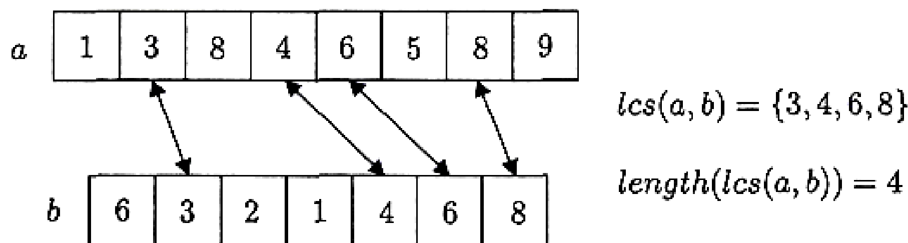
$$lcs[x][y] = \max\{d[x-1][c_1-1], \ldots, d[x-1][c_k-1]\} + \{a_x\}.$$

Како је $length(lcs[x][y_1]) \leq length(lcs[x][y_2])$ за $y_1 \leqslant y_2$ претходну формулу можемо написати као

$$lcs[x][y] = lcs[x-1][c_k-1] + \{a_x\},$$

где је c_k индекс последњег појављивања елемента a_x у поднизу b^y. Када сложимо претходну причу, $lcs[x][y]$ се рачуна на следећи начин:

$$lcs[x][y] = \max\{lcs[x-1][y], lcs[x-1][c_k-1] + \{a_x\}\}. \tag{2}$$



Слика 4: пример најдужег заједничког подниза

Покушајмо да опишемо сложеност алгоритма Ф. Како једини „проблем“ у формули представља проналажење самог индекса c_k, можемо написати да је сложеност $O(n \times m \times FindC_k)$. Индекс c_k можемо претраживати линеарно што би нас довело до сложености $O(nm^2)$.

Оставимо сада формулу (2) са стране и покушајмо да приступимо рачунању $lcs[x][y]$ на други начин. Уколико би елементи a_x и b_y били једнаки, тада важи $lcs[x][y] = lcs[x-1][y-1] + \{a_x\}$. Шта би било у случају да су они различити? Како се наведени елементи налазе на последњим позицијама низова a^x и b^y, у случају да оба улазе у НЗП, тада би морали бити једнаки, што је супротно претпоставци. Дакле, у том случају можемо тврдити да је $lcs[x][y] = \max\{lcs[x][y-1], lcs[x-1][y]\}$.

Горња релација нам омогућава да израчунамо све елементе $lcs[x][y]$ редом по врстама или по колонама, знајући да је $lcs[x][0] = lcs[0][y] = \emptyset$. Наравно, није потребно памтити саме подnizове. Довољно је памтити њихове дужине – означићемо их са $lenLCS[x][y]$. Реконстукција самог подниза се слично претходним проблемима може реализовати из матрице $lenLCS$ или памћењем претходника.

Сложеност новог алгоритма је $O(nm)$. Ова два алгоритма нам демонстрирају једну битну чињеницу. Наиме, није увек толико очигледно коју рекурзивну везу треба уочити. За исту дефиницију потпроблема можемо имати више веза, које као што видимо, не морају имати исте сложености. Овде чак имамо да је бољи алгоритам и далеко једноставнији за имплементацију.

Алгоритам Ф. Псеудо код проблема НЗП

Input: dva niza a i b dužina n odnosno m
Output: LCS - najduži zajednički podniz

```
1  for j ← 0 то m do
2  |  lenLCS[0][j] = 0;
3  endfor
4  for i ← 0 то n do
5  |  lenLCS[i][0] = 0;
6  endfor
7  for i ← 1 то n do
8  |  for j ← 1 то m do
9  |  |  if a[i] = b[j] then
10 |  |  |  lenLCS[i][j] = lenLCS[i − 1][j − 1] + 1;
11 |  |  else
12 |  |  |  lenLCS[i][j] = max{lenLCS[i][j − 1], lenLCS[i − 1][j]};
13 |  |  endif
14 |  endfor
15 endfor
16 LCS = ∅;
17 x = n; y = m;
18 while (x > 0 and y > 0) do
19 |  if a[x] = b[y] then
20 |  |  add a[x] to LCS;
21 |  |  x = x − 1; y = y − 1 ;
22 |  else
23 |  |  if lenLCS[x][y − 1] > lenLCS[x − 1][y] then
24 |  |  |  y = y − 1;
25 |  |  else
26 |  |  |  x = x − 1;
27 |  |  endif
28 |  endif
29 endw
30 return LCS;
```

Напомена. Уколико би се тражила само дужина НЗПа, тада можемо да уштедимо меморијски простор. Наиме, уместо целе матрице $lenLCS$ можемо да имамо само два низа, који би играли улогу врсте из $lenLCS$ која се тренутно формира и претходне врсте. Ово можемо урадити пошто нам израчунавање елемента $lenLCS[x][y]$ зависи само од елемената у врстама x и $x-1$.

У наредом број упознаћемо се са још пар интересантних проблема и приступа. Наравно биће дати и проблеми које ћемо оставити читаоцима за вежбу.

Увод у динамичко програмирање - наставак

***Андреја Илић*, Ниш**

Најдужи растући подниз

Проблем 1. [***Најдужи растући подниз (НРП)***] *Дат је низ a дужине n. Треба одредити најдужи подниз не обавезно узастопних елемената датог низа, који је растући.*

Приказаћемо три различита приступа решавању проблема најдужег растућег подниза (енг. Longest Increasing Subsequence). У дефиницији проблема је наведено да се тражи растући подниз, тако да ћемо ми у даљем тексту под растућим, без губљења општости, подразумевати строго растући. На примеру $a = \{1, 9, 3, 8, 11, 4, 5, 6, 4, 19, 7, 1, 7\}$ можемо видети да је најдужи растући подниз дужине 6 и то је $\{1, 3, 4, 5, 6, 7\}$.

Најједноставније решење јесте да сведемо овај проблем на већ познати проблем налажења најдужег заједничког подниза. Конструишемо низ b, који представља сортирани низ a у времену $O(n \log n)$, а затим нађемо најдужи заједнички подниз за низове a и b у времену $O(n^2)$.

Такође, постоји праволинијско решење динамичким програмирањем које је квадратне сложености. Иако су сложености еквивалентне, овај алгоритам је знатно бржи у пракси. Нека је $d[k]$ дужина најдужег растућег подниза низа a, са ограничењем да је последњи елемент управо $a[k]$. Глобални НРП се мора завршити на неком елементу низа a, па ћемо коначно решење добити налажењем максимума у низу d.

Остаје да рекурзивно израчунамо елементе низа d. Када рачунамо $d[k]$, посматрамо скуп свих индекса S_k за које важи $i < k$ и $a[i] < a[k]$. Ако је скуп S_k празан, тада су сви елементи који се налазе пре $a[k]$ у низу a већи од њега, што доводи до $d[k] = 1$. Иначе, ако максимизирамо дужину најдужег растућег подниза у оквиру скупа S_k, тада само додамо елемент $a[k]$ на крај овог низа. Закључујемо да важи следећа формула:

$$d[k] = \max\{d[i] \mid i \in S_k\} + 1 = \max_{i<k, a[i]<a[k]} d[i] + 1.$$

Уколико је потребно налажење једног НРП (пошто тражени подзниз не мора бити јединствен), то можемо урадити коришћењем помоћног низа p. Наиме, $p[k]$ ће нам представљати индекс i, који је максимизирао горњи израз при рачунању $d[k]$. Такође, најдужи растући подниз се може реконструисати и једним проласком кроз низ уназад.

Сада ћемо приказати решење у временској сложености $O(n \cdot \log k)$, где је k дужина најдужег растућег подниза. Дефинишимо $A_{i,j}$ као:

$$A_{i,j} = \text{најмањи могући последњи елемент свих растућих низова дужине } j \text{ користећи елементе } a[1], a[2], \ldots, a[i].$$

Алгоритам Г. Псеудо код проблема НРП

```
Input: Низ a дужине n
Output: LIS - најдужи растући подниз низ a
1  for k ← 1 to n do
2  |  max = 0;
3  |  for i ← 1 to k − 1 do
4  |  |  if a[k] > a[i] and d[i] > max then
5  |  |  |  max = d[j];
6  |  |  endif
7  |  endfor
8  |  d[k] = max + 1;
9  endfor
10 max = 0;
11 index = 0;
12 for k ← 1 to n do
13 |  if d[k] > max then
14 |  |  max = d[k];
15 |  |  index = k;
16 |  endif
17 endfor
18 LIS ← ∅;
19 while d[index] > 1 do
20 |  add a[index] to LIS;
21 |  i = index − 1;
22 |  while d[index] > d[i] + 1 do
23 |  |  i = i − 1;
24 |  endw
25 |  index = i;
26 endw
27 add a[index] to LIS;
28 return LIS;
```

Приметимо да за свако i, важи

$$A_{i,1} < A_{i,2} < \cdots < A_{i,j}.$$

Дакле, када тражимо најдужи растући подниз који се завршава елементом $a[i+1]$ потребно је да нађемо индекс j такав да је $A_{i,j} < a[i+1] \leqslant A_{i,j+1}$. У том случају ће тражена дужина бити једнака $j+1$ и $A_{i+1,j+1}$ ће бити једнако a_{i+1}, док ће остали елементи низа A_{i+1} остати непромењени. Закључујемо да постоји највише једна разлика између низова A_i и A_{i+1}. Како је низ A_i увек сортиран и после примене горње операције - можемо користити **бинарну претрагу** (енг. binary search).

Алгоритам X. Псеудо код проблема НРП

Input: Низ a дужине n
Output: LIS - најдужи растући подниз

```
1  max = 1;
2  LIS[1] = 0;
3  for i ← 1 to n do
4  |   if a[LIS[max]] < a[i] then
5  |   |   p[i] = LIS[max];
6  |   |   max = max + 1;
7  |   |   LIS[max] = i;
8  |   endif
9  |   else
10 |   |   left = 0;
11 |   |   right = max − 1;
12 |   |   while (left < right) do
13 |   |   |   m = (left + right)/2;
14 |   |   |   if a[LIS[m]] < a[i] then
15 |   |   |   |   left = m + 1;
16 |   |   |   else
17 |   |   |   |   right = m;
18 |   |   |   endif
19 |   |   endw
20 |   |   if a[i] < a[LIS[left]] then
21 |   |   |   if left > 0 then
22 |   |   |   |   p[i] = LIS[left − 1];
23 |   |   |   endif
24 |   |   |   LIS[left] = i;
25 |   |   endif
26 |   endif
27 endfor
28 i = LIS[max];
29 for k ← max to 1 do
30 |   i = p[i];
31 |   LIS[k] = i;
32 endfor
33 return LIS;
```

Проблем ранца

Проблем ранца (енг. Knapsack problem) је један од најпознатијих проблема ДП. Једна од једноставнијих варијанти проблема ранца је наредни пример, који ће нас лепо увести у суштину основне формулације проблема.

Проблем 2. *Дат је низ природних бројева a дужине n и природни број $S \leq 10^5$. Пронаћи подниз низа a чија је сума једнака S или установити да такав подниз не постоји.*

Видимо да у поставци проблема постоји ограничење за број S. Као што ћемо ускоро видети, ово нам је потребно пошто ћемо пратити нека међустања којих ће бити управо S. За почетак покушајмо да установимо постојање подниза, а потом ћемо видети како га и пронаћи. Низ a дужине n има тачно 2^n поднизова – па испитивање свих поднизова одмах одбацујемо као неефикасно решење. Дефинишимо зато низ sum дужине S на следећи начин:

$$sum[k] = \begin{cases} true, & \text{уколико постоји подниз са сумом } k, \\ false, & \text{иначе.} \end{cases}$$

Низ sum ћемо рачунати рекурзивно (додаваћемо један по један елемент низа a и ажурирати вредности низа sum).

- На почетку све елементе низа sum иницијализујемо на $false$, осим $sum[0]$ који ће бити $true$. Ово одговара случају када из низа a нисмо узели ниједан елемент.

- Претпоставимо да смо до сада додали елементе $a[1], \ldots, a[k-1]$ (тј. подниз a^{k-1}) и желимо да додамо елемент са индексом k. Које елементе низа sum треба променити? Уколико је постојао подниз низа a^{k-1} који је имао суму s, тада је тај исти подниз и подниз низа a^k, па вредност $sum[k]$ треба остати непромењена и једнака $true$. Међутим, додавањем новог елемента постоји и подниз чија је сума $s + a[k]$ коју треба поставити на $true$ (уколико је она једнака $false$ и важи $s + a[k] \leqslant S$).

Одговор на питање да ли постоји подниз са сумом S наћи ћемо у вредности елемента $sum[S]$. Уколико нас занима и сам подниз, памтићемо за сваки елемент $sum[k]$ преко ког елемента низа a смо дошли. Тачније, увек када неки елемент низа sum поставимо на $true$, елемент из друге ставке $a[k]$ памтимо као претходника. Као претходника елемента $sum[0]$ као и све остале до којих нисмо успели да дођемо, поставићемо на -1. На тај начин, једноставном итерацијом по претходницима елемента $sum[S]$, све док тај претходник не постане -1, добијамо тражени подниз.

Напомена. Лако можемо приметити да наведени подниз није јединствен. Уколико бисмо желели да реконструишемо све поднизове, треба памтити све претходнике или их рачунати при самој реконструкцији (k-ти елемент је претходник за елемент $sum[s]$ уколико важи $sum[s - a[k]] = true$).

Напомена. Посебно пажњу треба обратити на корак 7 у Алгоритму И, где смо низ sum обилазили од назад. У супротном бисмо имали случај да сваки елемент низа можемо употребити произвољан број пута. Наиме, уколико би низ a био састављен само од једног елемента, наш низ sum био имао вредност $true$ на позицијама $0, a[0], 2a[0], \ldots$. Разлог за

Алгоритам И. Псеудо код проблема подниза задате суме

```
    Input: Низ a дужине n; сума S коју испитујемо
    Output: subArray - подниз са сумом S (празан уколико такав не постоји)
 1  for i ← 0 to S do
 2  |   sum[i] = false;
 3  |   prior[i] = −1;
 4  endfor
 5  sum[0] = true;
 6  for k ← 1 to n do
 7  |   for s ← S downTo 0 do
 8  |   |   if sum[s] and sum[s] + a[k] ≤ S then
 9  |   |   |   sum[s + a[k]] = true;
10  |   |   |   prior[s + a[k]] = k;
11  |   |   endif
12  |   endfor
13  endfor
14  subArray = ∅;
15  if sum[S] then
16  |   currentSum = S;
17  |   while prior[currentSum] ≠ −1 do
18  |   |   add prior[currentSum] to subArray;
19  |   |   currentSum = currentSum − a[prior[currentSum]];
20  |   endw
21  endif
22  return subArray;
```

ово је једноставан: $sum[a[0]]$ ћемо маркирати преко $sum[0]$. Каснијом итерацијом долазимо на елемент $sum[a[0]]$ који је $true$ због чега маркирамо елемент $sum[a[0] + a[0]]$ итд.

Као што смо напоменули, проблем ранца има неколико варијанти. Дајемо ону формулацију по којој је проблем и добио име.

Проблем 3. *Провалник са ранцем запремине N упао је у просторију у којој се чувају вредни предмети. У просторији има укупно M предмета. За сваки предмет позната је његова вредност $v[k]$ и његова запремина $z[k]$, $k \in [1, M]$. Све наведене вредности су целобројне. Провалник жели да напуни ранац највреднијим садржајем. Потребно је одредити које предмете треба ставити у ранац.*

Дакле, треба изабрати оне предмете за које је сума запремина мања или једнака N, а чија је сума вредности максимална. На почетку прокоментаришимо неколико карактеристика проблема.

- Ранац који је оптимално попуњен не мора бити попуњен до врха (у неким случајевима то и неће бити могуће). На пример, посматрајмо случај у коме је $N = 7$ а $v = \{3, 4, 8\}$, $z = \{3, 4, 5\}$. Овде је оптимално решење узети само последњи предмет чиме би вредност ранца била 8, док ранац не би био у потпуности попуњен.

- Највреднији предмет не мора ући у решење: идеја да предмете треба сортирати по „вредности по јединици запремине“, тј. по $v[k]/z[k]$ није коректна. Овај приступ (грабљиви метод) не даје увек оптимално решење, што показује пример: $N = 7$ и $v = \{3, 4, 6\}, z = \{3, 4, 5\}$. Овде бисмо грабљивим алгоритмом као решење узели 3. предмет, док се оптимално решење представљају предмети 1 и 2.

Из ове дискусије се види да проблем није једноставан и да се до решења не може доћи директно. Анализирајући проблем, можемо приметити следеће: ако је при оптималном попуњавању ранца последњи изабрани предмет k, онда преостали предмети представљају оптимално попуњавање ранца запремине $N - z[k]$. Ова констатација се лако доказује свођењем на контрадикцију (као и код већине обрађених проблема). Према томе, оптимално попуњавање ранца садржи оптимално попуњавање мањег ранца, што нас наводи на ДП. Дефинишимо низ $d[k]$ за $k \in [1, N]$ као:

$d[k]$ = максимална вредност ранца запремине k при чему је ранац попуњен до врха.

Алгоритам 1. Псеудо код проблема ранца

Input: Низови z и v дужине M; запремина ранца N
Output: *knapsack* - низ оптималних предмета

```
1  for i ← 1 to S do
2  |   d[i] = −∞;
3  |   prior[i] = −1;
4  endfor
5  d[0] = 0;
6  for k ← 1 to M do
7  |   for i ← N − z[k] downTo 0 do
8  |   |   if d[i] ≠ −∞ and d[i + z[k]] < d[i] + v[k] then
9  |   |   |   d[i + z[k]] = d[i] + v[k];
10 |   |   |   prior[i + z[k]] = k;
11 |   |   endif
12 |   endfor
13 endfor
14 knapsack = ∅;
15 index = max{i|i ∈ [0, N] and d[i] ≠ −∞};
16 while prior[index] ≠ −1 do
17 |   currentItem = prior[index];
18 |   адд currentItem то knapsack;
19 |   index = prior[index − z[currentItem]];
20 endw
21 return knapsack;
```

Оптимална вредност ранца запремине N се налази у неком од елемената $d[k]$ за $k \in [1, N]$. Из горњег разматрања видимо да се рачунање вредности $d[m]$ може свести на рачунање елемента $d[m - z[k]]$ за неки предмет k. Слично као и код претходног проблема, низ d можемо рачунати додавањем једног по једног предмета.

На почетку имамо случај празног ранца и $d[0] = 0$ као базу. Како се тражи максимална вредност ранца, можемо све вредности низа d, осим позиције 0, поставити на $-\infty$ (на крају алгоритма уколико је $d[k] = -\infty$, то ће значити да датим предметима не можемо попунити у потпуности ранац запремине k). Претпоставимо сада да смо додали првих $k-1$ предмета и желимо да додамо и k-ти предмет. Питамо се шта би било да ранац садржи предмет k. Уколико би са њим ранац имао запремину $S \leqslant N$ тада би ранац запремине $S - z[k]$ био попуњен неким подскупом првих $k-1$ предмета. Зато ћемо сваки елемент $d[s] \neq -\infty$ покушати да проширимо за нови предмет. Дакле, испитујемо да ли је $d[s+z[k]] < d[s]+v[k]$ и уколико јесте мењамо вредност $d[s+z[k]]$ на оптималнију вредност за ову запремину – $d[s]+v[k]$.

Да бисмо реконструисали сам низ предмета које ћемо стављати у ранац, за сваку вредност $d[k]$ ћемо памтити индекс последњег додатог предмета (у низу $prior$). Псеудо код горе описаног алгоритма је приказан у Алгоритму J.

Разни задаци

Даћемо сада неколико задатака, а решења ће бити објављена у наредном броју Тангенте.

Задатак 1. [Србија, Окружно такмичење, 2000] *Дата је бинарна матрица димензије $n \times n$, $n \leqslant 1000$. Нађи димензију највеће квадратне подматрице која је састављена само од нула.*

Задатак 2. [TopCoder, SRM 255] *Дати су бројеви $0 \leqslant M < N \leqslant 1.000.000$. Одредити најмањи број састављен само од непарних цифара, који при дељењу са N даје остатак M. Уколико тражени број не постоји, штампати -1.*

Задатак 3. [Србија, Републичко такмичење 2003] *Дат је низ различитих природних бројева дужине n. Над низом можете извршавати две операције: произвољан елемент из низа пребаците на његов почетак или на његов крај. Одредити минимални број операција, тако да резултујући низ буде растући.*

Задатак 4. [ACM SPOJ, PERMUT1] *Дата су два природна броја $n \leqslant 1000$ и $k \leqslant 10000$. Нађи број пермутација бројева од 1 до n који имају тачно k инверзија. Број инверзија у пермутацији $(p_1, p_2, \ldots, p_n)$ је број парова (i, j) таквих да је $p_i > p_j$ за $1 \leqslant i < j \leqslant n$.*

Задатак 5. [Хрватска, Изборно 2001] *Дат је низ d реалних бројева дужине $n \leq 1000$, који представљају удаљеност ноћних светиљки од почетка улице. За сваку од светиљки је дата и њена потрошња у јединици времена када је упаљена. Перица се на почетку налази на растојању $start$ од почетка улице и жели да погаси све светиљке у улици. Он се креће брзином од једног метра у јединици времена. Колика је минимална потрошња светиљки које оне произведу - док их Перица све не погаси?*

Решења задатака из динамичког програмирања

***Андреја Илић*, Ниш**

Задатак 1. [Србија, Окружно такмичење, 2000] *Дата је бинарна матрица димензије $n \times n$, $n \leqslant 1000$. Нађи димензију највеће квадратне подматрице која је састављена само од нула.*

Решење. Сваку квадратну подматрицу можемо дефинисати њеном величином и једним теменом, на пример, доњим десним. Покушајмо да за свако поље матрице израчунамо највећу димензију подматрице састављене само од нула, тако да је дато поље управо доњи десни десни угао. Означимо тражену величину са $d[x][y]$. Уколико би $d[x][y]$ било једнако k, тада би $d[x-1][y]$ морало бити најмање $k-1$, баш као и $d[x][y-1]$. Међутим, ово није и довољан услов, јер нам он ништа не говори о вредности у горњем левом пољу наше подматрице димензије k. Тај услов можемо проверити преко $d[x-1][y-1] \geqslant k-1$. Дакле, добијамо

$$d[x][y] = \begin{cases} 0, & \text{ако је } matrix[x][y] = 1, \\ 1 + \min\{d[x-1][y], d[x][y-1], d[x-1][y-1]\}, & \text{ако је } matrix[x][y] = 0, \end{cases}$$

где, наравно, морамо пазити да наведени индекси припадају матрици. На почетку можемо израчунати вредности у првој врсти и колони или додатни вештачке 0-те врсте и колоне. Уколико елементе матрице d рачунамо кретањем по врстама са лева на десно у тренутку рачунања елемента на позицији (x, y) потребне вредности са десне стране једнакости ће бити већ израчунате.

0	1	0	0	0
0	0	0	1	0
0	0	0	0	1
1	0	0	0	0
0	0	0	0	0

Слика 1. Пример највеће квадратне подматрице чији су елементи само 0.

Задатак 2. [TopCoder, SRM 255] *Дати су бројеви $0 \leqslant M < N \leqslant 1.000.000$. Одредити најмањи број састављен само од непарних цифара, који при дељењу са N даје остатак M. Уколико тражени број не постоји, штампати -1.*

Решење. Како конструисати све бројеве састављене од непарних цифара? Постоји тачно пет таквих једноцифрених бројева: 1, 3, 5, 7 и 9. Да би добили бројеве са L непарних цифара, довољно је помножити сваки од $(L-1)$–цифрених бројева састављених од непарних цифара са 10 и додати сваку од непарних цифара.

За решење ће нам бити потребан један низ q. На почетку убацимо све једноцифрене непарне бројеве у q. Затим додајемо све непарноцифрене бројеве који су добијени од 1,

затим све непарноцифрене бројеве добијене од 3 и тако даље. Тиме ћемо конструисати све непарноцифрене бројеве у растућем редоследу. Наравно, све бројеве узимамо по модулу N.

Нема смисла имати два броја са једнаким остатком у низу q, пошто ће они генерисати исте непарноцифрене бројеве. Зато користимо низ d дужине N. Наиме, $d[i]$ представља најмањи број који при дељењу са N даје остатак i. На почетку, иницијализујемо елементе низа d на -1. Можемо приметити да не морамо памтити целе бројеве (пошто могу бити састављени од много цифара), већ је довољно памтити последњу цифру и претходника.

Меморијска и временска сложеност овог алгоритма је $O(N)$.

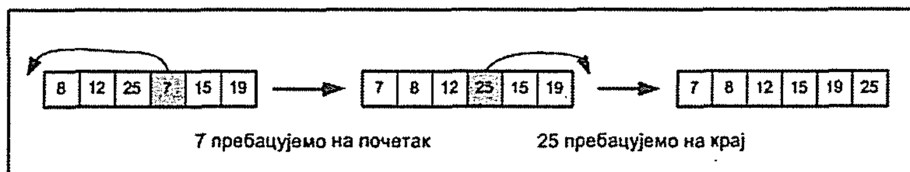
Задатак 3. [Србија, Републичко такмичење 2003] *Дат је низ различитих природних бројева дужине n. Над низом можете извршавати две операције: произвољан елемент из низа пребаците на његов почетак или на његов крај. Одредити минимални број операција, тако да резултујући низ буде растући.*

Решење. Означимо дати низ са a и претпоставимо да смо при симулацији сортирања померили елементе са индексима $1 \leqslant x_1 < \cdots < x_k \leqslant n$. Елементи које нисмо преместили задржали су свој поредак као и на почетку. Како на крају треба добити сортирани низ, поредак непремештаних елемената мора бити сортиран. Ова чињеница нас наводи на то да треба пронаћи најдужи растући подниз низа. Међутим, приметимо да се ниједан нови елемент не може убацити између нека два елемента која нисмо премештали. Зато налазимо најдужи растући подниз низа a узастопних елемената (не по индексима већ по вредностима).

На почетку, ради лакше манипулације, можемо елементе низа пресликати у пермутацију бројева од 1 до n. Низ d дефинишимо као:

$$d[n] = \text{најдужи растући подниз елемената са индексима из сегмента } [1, n]$$
$$\text{који се завршава } n\text{–тим елементом.}$$

Низ d можемо рачунати линеарно, за разлику од најдужег растућег подниза, јер овде знамо који елемент једини може да буде претходник (претходник за $d[k]$ је елемент који има вредност $a[k] - 1$).



Слика 2. Пример сортирања у Задатку 3.

Задатак 4. [ACM SPOJ, PERMUT1] *Дата су два природна броја $n \leqslant 1000$ и $k \leqslant 10000$. Наћи број пермутација бројева од 1 до n који имају тачно k инверзија. Број инверзија у пермутацији $(p_1, p_2, \ldots, p_n)$ је број парова (i, j) таквих да је $p_i > p_j$ за $1 \leqslant i < j \leqslant n$.*

Решење. Стање ћемо описати са два параметра: дужином пермутације и бројем инверзија. Дакле, означимо са

$$d[i][j] = \text{број пермутација дужине } i \text{ са } j \text{ инверзија.}$$

Посматрајмо јединицу у тим пермутацијама и њену улогу у броју инверзија. Уколико се она налази на првом месту, онда она не учествује ни у једној инверзији, па тих пермутација

има $d[i-1][j]$ (јер је свеједно да ли гледамо пермутације бројева $(1,2,\ldots,n-1)$ или $(2,3,\ldots,n)$). Ако се јединица налази на другом месту, тада пермутација са j инверзија има $d[i-1][j-1]$. На овај начин долазимо до рекурентне формуле:

$$d[n][k]=\sum_{i=0}^{n-1} d[n-1,k-i].$$

Покушајмо да наведена поља матрице d рачунамо на оптималан начин:

$$d[n][k]=d[n][k-1]+d[n-1][k]-d[n-1][k-n].$$

Горњу релацију није тешко приметити. Наиме, ако се напишу суме за елементе $d[n][k]$ и $d[n][k-1]$, види се да се оне разликују само у два сабирака. Ово додатно смањује како временску, тако и меморијску сложеност. Елементе матрице d рачунамо у $O(1)$, дакле временска сложеност алгоритма је $O(nk)$ док је меморијска $O(n)$, јер памтимо само последња два реда матрице. За почетне вредности је довољно узети $d[i][0]=1$ и $d[0][j]=0$. Напоменимо да треба водити рачуна о томе да ли је број инверзија k већи од бројача i и броја елемената n.

Задатак 5. [Хрватска, Изборно 2001] *Дат је низ* d *реалних бројева дужине* $n \leqslant 1000$*, који представљају удаљености ноћних светиљки од почетка улице. За сваку од светиљки је дата и њена потрошња у јединици времена када је упаљена. Перица се на почетку налази на растојању* $start$ *од почетка улице и жели да погаси све светиљке у улици. Он се креће брзином од једног метра у јединици времена. Колика је минимална потрошња светиљки које оне произведу - док их Перица све не погаси?*

Решење. Означимо потрошњу k-те сијалице са $w[k]$. На почетку ради лакшег рачунања сортирајмо сијалице по растојањима и трансформишимо координате тако да се Перица налази на месту са координатом 0. Убацимо и једну "вештачку сијалицу" на координати 0. Дефинишимо следеће ознаке:

$L[x][y]=$ минимална потрошња потребна да се погасе сијалице са индексима $x,x+1,\ldots,y$ и да се на крају Перица налази код сијалице x,

$D[x][y]=$ минимална потрошња потребна да се погасе сијалице са индексима $x,x+1,\ldots,y$ и да се на крају Перица налази код сијалице y,

$T[x][y]=$ укупна снага коју троше сијалице са индексима $x,x+1,\ldots,y$,

$T^c[x][y]=$ укупна снага коју троше сијалице са индексима $1,\ldots,x-1,y+1,\ldots,n$.

Елементи помоћних матрице T и T^c ће нам бити потребне за рачунање главних матрица L и D у којима се садржи и крајње решење $\min\{L[1][n],D[1][n]\}$. Разлог за тако дефинисане везе јесте чињеница да уколико Перица погаси сијалице a и b, тада ће погасити и сијалице са индексима између a и b.

Погледајмо прво како можемо елегантно израчунати матрицу T, пошто ћемо сличну идеју касније користити за матрице D и L. Помоћу једнакости

$$T[x][y]=w[x]+T[x+1][y],$$

рачунање неке вредности за сегмент $[x, y]$ дијаматра $y - x$ можемо да сведемо на рачунање сегмента $[x+1, y]$ који је дијаматра $y - x - 1$. Уколико би елементе матрице T рачунали на такав начин да су у тренутку рачунања поља дијаметра k елементи са мањим дијаметрима иницијализовани, горњу једнакост би могли имплементирати и без рекурзије. Идеја је да се матрица T попуњава дијагонално.

Алгоритам К. Псеудо код рачунања матрице T из задатка 5

```
   Input: Низ потрошња сијалица w дужине n
   Output: матрица T
1  for k ← 1 to n do
2  |  T[k][k] = w[k];
3  endfor
4  for diameter ← 2 to n − 1 do
5  |  for x ← 1 to n − diameter do
6  |  |  y = x + diameter − 1;
7  |  |  T[x][y] = w[x] + T[x + 1][y];
8  |  endfor
9  endfor
10 return T;
```

Елементе матрице T^c рачунамо као

$$T^c[x][y] = S - T[x][y], \quad \text{где је } S = \sum_{k=1}^{n} w[k].$$

Сада можемо дефинисати рекурентне везе за матрице L и D:

$$L[x][y] = \min \begin{cases} L[x+1][y] + T^c[x+1][y] \cdot (d[x+1] - d[x]), \\ D[x+1][y] + T^c[x+1][y] \cdot (d[y] - d[x]), \end{cases}$$

$$D[x][y] = \min \begin{cases} D[x][y-1] + T^c[x][y-1] \cdot (d[y] - d[y-1]), \\ L[x][y-1] + T^c[x][y-1] \cdot (d[y] - d[x]). \end{cases}$$

Објаснимо рекурентну релацију за $L[x][y]$, пошто се веза за матрицу D добија аналогно. Наиме, да би Перица погасио сијалице из сегмента $[x, y]$ на крају мора бити или код сијалице x или код сијалице y. Како би угасио сијалице $[x, y]$, мора пре тога угасити сијалице $[x-1, y]$. За њихово гашење постоје два случаја: да се налази у $x - 1$ или y, што уједно и представљају горње случајеве за рачунање елемента $L[x][y]$.

Статијата прв пат е објавена во списанието ТАНГЕНТА на ДМ на Србија во 2010/11 година