Семинарски рад из предмета
Алгоритми и структуре података,

Индустрија 4.0, Математички факултет, Универзитет у Београду

Бектрекинг

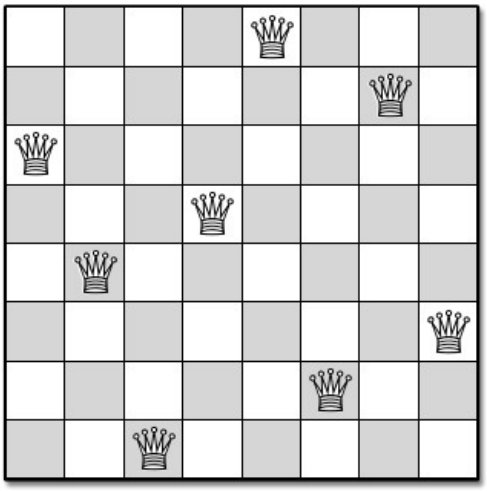
Студент: Стефан Ковач 4003/2020
Професор: Весна Маринковић
Асистент: Сана Стојановић Ђурђевић

У овом раду се описује једна важна рекурзивна стратегија која се назива претрага са повратком или бектрекинг (eng. *backtracking*). Бектрекинг алгоритам покушава да конструише решење рачунарског проблема постепено, део по део. Кад год алгоритам треба да се одлучи између више алтернатива, које могу бити наредне компоненте решења, он рекурзивно обрађује сваку алтернативу, а затим бира најбољу.

У тексту који следи размотрићемо неколико типичних проблема који се могу решити техником претраге са повратком и установити општи образац за такве проблеме. Такође ћемо урадити неколико задатака применом технике претраге са повратком и решења ћемо имплементирати у програмском језику *C++*.

1.1 *N дама*

Основни бектрекинг проблем је проблем ***n* дама**. Први га је представио немачки љубитељ шаха Макс Безел (Max Bezzel) 1848. године за стандардну таблу димензија *8 x 8* и Франсоа-Жозеф Есташ Лионе (François-Joseph Eustache Lionnet) 1869. године за општију *n x n* таблу. Проблем гласи: поставити *n* дама на *n x n* шаховску таблу, тако да се ниједне две даме међусобно не нападају. За читаоце који нису упознати са правилима шаха, то значи да се никоје две даме не налазе у истом реду, у истој колони или на истој дијагонали.



Слика 1 - Гаусово прво решење за проблем 8 дама, представљено као низ *[5,7,1,4,2,8,6,3]* (n-ти елемент у низу означава које поље по реду је дама заузела у n-том реду)

У писму свом пријатељу Хајнриху Шумахеру (Heinrich Schumacher) 1850. године, угледни математичар Карл Фридрих Гаус (Carl Friedrich Gauss) написао је да се Франц Наукова тврдња, да проблем осам дама има 92 решења, може лако потврдити методом покушаја и грешака за неколико сати.

Гаус је у писму описао следећу рекурзивну стратегију за решавање проблема *n дама:* Постављамо даме на таблу ред по ред, почев од горњег реда. Да бисмо поставили *r*-ту даму, испробавамо свих *n* поља у реду *r* слева надесно у једноставној *for* петљи. Ако је одређено поље нападнуто од неке од претходно постављених дама, прескачемо то поље. У супротном постављамо даму на то поље и рекурзивно покушавамо да поставимо преостале даме у наредне редове.

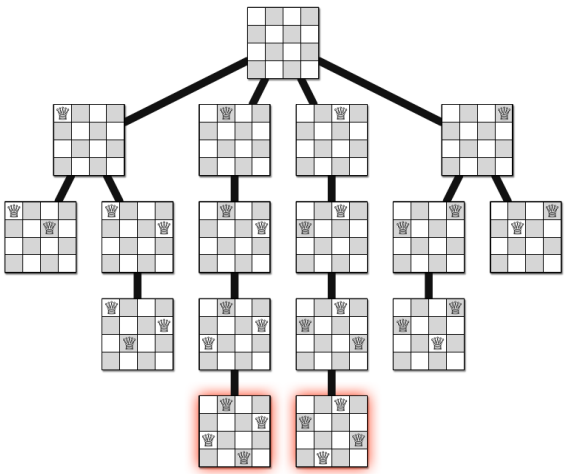
У псеудокоду 1 приказан је резултујући алгоритам, који рекурзивно набраја сва решења проблема *n* дама. Следећи Гауса, позиције дама на шаховској табли представљају се помоћу низа *Q* дужине *n*, где *Q[i] (i*-ти елемент низа *Q)* означава које поље у реду *i* садржи даму. Приликом позива

функције *PostaviDame* улазни параметар *r* је индекс првог празног реда, а префикс *Q[1..r-1]* (подниз низа *Q*, почевши од позиције *1* до позиције *r-1*) садржи позиције првих *r-1* дама. Спољашња *for* петља разматра постављање даме на свако од поља у реду *r*. Унутрашња *for* петља проверава да ли је то постављање у складу са већ постављеним дамама у претходних *r-1* редова. Уколико јесте, поставља се дама на то поље, и рекурзивно се позива функција за постављање даме на следећи ред. Ако је *r = n+1* то значи да смо успешно поставили даме на све редове, уколико смо почели да их постављамо од првог реда, па исписујемо то решење. Да бисмо одредили сва исправна постављања *n* дама на таблу димензија *n* x *n* потребно је позвати функцију *PostaviDame(Q[1..n],1)*.

```
PostaviDame(Q[1..n],r):
    ako r = n + 1
        ispiši Q[1..n]
    inače
        za j od 1 do n
            legalan_potez = Tačno
            za i od 1 do r-1
                ako (Q[i] = j) ili (Q[i] = j + r – i) ili (Q[i] = j – r +i)
                    legalan_potez = Netačno
            ako legalan_potez
                Q[r] = j
                PostaviDame(Q[1..n], r+1)    <<Rekurzija>>
```

Псеудокод 1 - Гаусов бектрекинг алгоритам за проблем *n* дама

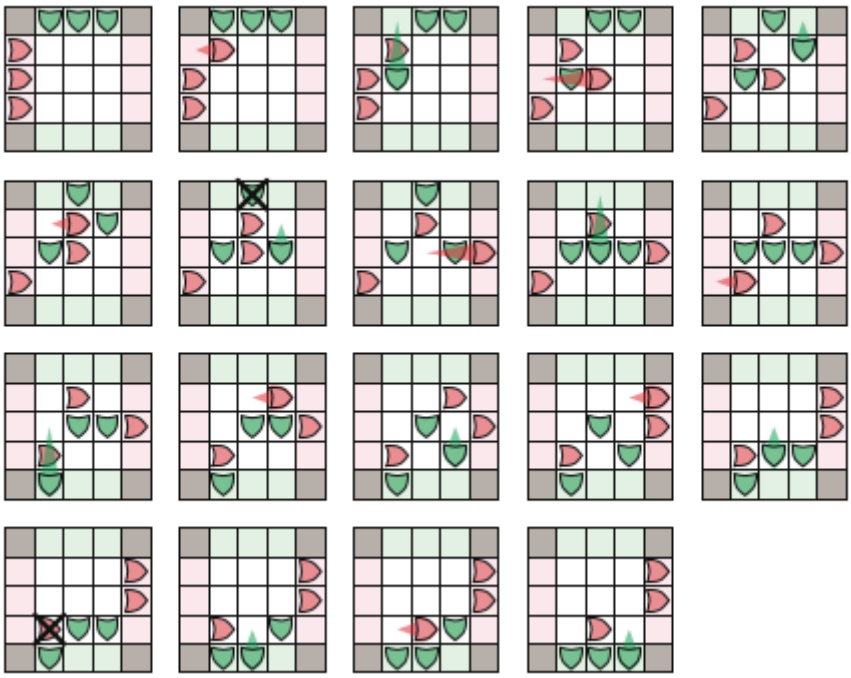
Извршење функције *PostaviDame* може се илустровати помоћу стабла рекурзивних позива (Слика 2). Сваки чвор у овом стаблу одговара потпроблему у којем су постављене даме у првих *r* редова, а проблем је пронаћи сва легитимна постављања дама у осталих празних *n-r* редова (r означава дубину у стаблу на којој се налази чвор, а n величину шаховске табле). Тиме сваки чвор одговара и парцијалном решењу. Специјално, корен одговара празној табли (са *r = 0*). Гране у стаблу рекурзивних позива одговарају рекурзивним позивима. Листови одговарају делимичним решењима која се не могу даље проширити, било због тога што у сваком реду већ постоји дама или зато што је свака позиција у следећем празном реду нападнута од већ постављених дама. Одвијање бектрекинг претраге свих решења еквивалентна је претраживању овог стабла по дубини.



Слика 2 – Комплетно стабло претраге Гаусовог алгоритма за проблем 4 даме

1.2 Стабла игре

Размотримо следећу једноставну игру за два играча која се игра на *n x n* квадратној табли са граничним пољима. На левим и горњим граничним пољима су почетне позиције жетона, док су на десним и доњим граничним пољима циљне позиције жетона (Слика 3). Играчи који играју ову игру се зову Лаза и Вера. Сваки играч има *n* жетона са којима се креће по табли с једне стране на другу. Лазини жетони започињу своје кретање са леве границе, по један у сваком реду и померају се водоравно удесно. Слично, Верини жетони започињу своје кретање са горње границе, по један у свакој колони и померају се вертикално надоле. Играчи повлаче потезе наизменично. У сваком свом потезу Лаза помери један од својих жетона удесно на празно поље или прескочи једним његовим жетоном један од Вериних жетона на празно поље два корака удесно. Ако неки од ова два потеза није могућ, Лаза прескаче потез и следећа игра Вера. Аналогно, ова правила важе и за Веру. Први играч који премести све своје жетоне на гранична поља на супротној страни табле побеђује. Није тешко доказати да све док су жетони на табли, бар један играч може легално да се креће и због тога сигурно неко на крају побеђује. Ако претпоставимо да један играч не може да направи легалан потез, то значи да два жетона другог играча блокирају кретање једног његовог жетона, док се његова друга два жетона налазе на циљним позицијама. Тада сигурно неки од ова два жетона који блокирају пут жетону првог играча, могу да направе легалан потез.

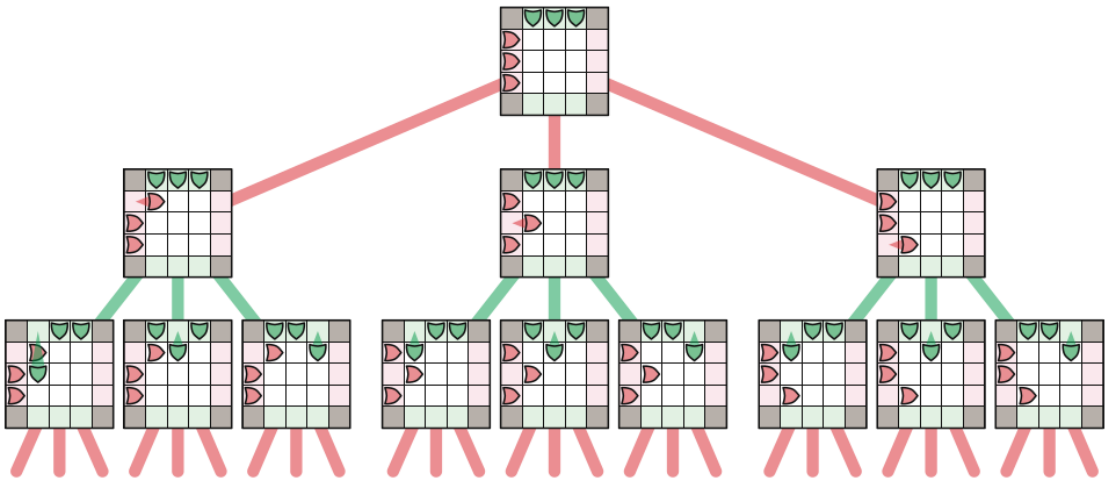


Слика 3 – Пример: Вера побеђује у *3 x 3* варијанти игре

Ако ову игру нисте већ видели, врло вероватно на први поглед нећете имати идеју како одиграти добро. Ипак, постоји релативно једноставан бектрекинг алгоритам који може да одигра оптимално ову или било коју другу игру за два играча, уколико у њој нема насумичности или скривености информација и која се завршава након коначног броја потеза. Односно, ако се нађете у неком стању игре (било почетно или не) и могуће је победити

против другог играча који игра оптимално, алгоритам ће вам рећи како да га победите.

Стање игре се састоји од положаја свих жетона на табли и идентитета играча који је на потезу. Ова стања се могу повезати у **стабло игре** (Слика 4), које има грану између стања *x* и *y* ако и само ако играч на потезу у стању *x* може легално да пређе у стање *y*. Корен стабла је почетно стање игре и свака путања од корена до листа одговара комплетној партији.



Слика 4 - Прва два нивоа стабла игре

Како би се кретало кроз ово стабло игре, рекурзивно се дефинише стање игре као добро или лоше:

- Стање је добро ако је играч на потезу победио или ако играч на потезу може да одигра потез и пређе у стање које је лоше за противничког играча.
- Стање је лоше ако је играч на потезу изгубио или ако сваки могућ потез води у стање које је добро за противничког играча

Еквивалентно, сваки чвор у стаблу који није лист је добар ако има једно лоше дете и лош је ако су му сва деца добра. Сваки играч који се нађе у добром стању у свом потезу може да победи, чак иако његов противник игра оптимално. То важи зато што играч који је у добром стању, а није победио, може да одигра потез који води противничког играча у лоше стање, а то значи да шта год одиграо противник његa ће довести опет у добро стање. Играч онда може да вуче потезе на овај начин све док не победи. С друге стране, ако се играч нађе у лошем стању игре, може победити само ако противник погреши, јер претходно наведеном тактиком противнички играч може сигурно да победи. Ова рекурзивна дефиниција предложена је од стране Ернста Зернела 1913. године.

Рекурзивна дефиниција одмах сугерише следећи рекурзивни бектрекинг алгоритам (Псеудокод 2) који одређује да ли је задато стање игре добро или лоше. У суштини, овај алгоритам је само претрага стабла по дубини. Еквивалентно, стабло игре је стабло рекурзивних позива алгоритма. Једноставна модификација овог бектрекинг алгоритма проналази добар потез (или чак све могуће добре потезе), ако је улаз стање игре које је добро.

```
OcenaStanja(X, igrač):
        ako je igrač pobednik u stanju X
                vrati Dobro
        ako je igrač gubitnik u stanju X
                vrati Loše
        za sve legalne poteze X → Y
                ako OcenaStanja(Y, !igrač) = Loše   <<!igrač je poritvnički igrač igrača igrač>>
                        vrati Dobro                 <<X → Y je dobar potez>>
        vrati Loše                                  <<Nema dobrih poteza iz X>>
```

Псеудокод 2 – Алгоритам који рачуна да ли је стање игре добро или лоше за играча који је на потезу

Сви програми за играње игара засновани су на овој једноставној бектрекинг стратегији. Међутим, пошто већина игара има огроман број стања, у пракси није могуће обићи цело стабло игре. Уместо тога, програми за игре користе разне хеуристике за поткресивање стабла игре. Они занемарују стања за која знају да су сигурно добра или лоша или барем боља или лошија од других стања. Такође, поткресују стабла на одређеној дубини и користе ефикасније хеуристике за процену листова.

1.3 Сума подскупа

Размотримо мало компликованији проблем звани **сума подскупа**. Дат је скуп X природних бројева и *циљни* број T. Питање је, да ли постоји подскуп од X тако да је сума његових елемената једнака T?

Приметимо да може постојати више таквих подскупова. На пример, ако је $X = \{8,6,7,5,3,10,9\}$ и $T = 15$, такав подскуп постоји. Такви подскупови су $\{8,7\}$, $\{7,5,3\}$, $\{6,9\}$ и $\{5,10\}$. С друге стране, ако је $X = \{11,6,5,1,7,13,12\}$ и $T = 15$, не постоји такав подскуп.

Постоје два тривијална случаја. Ако је циљни број $T = 0$, можемо одмах рећи да постоји тражени подскуп, јер је празан скуп подскуп сваког скупа X и сума његових елемената је 0. С друге стране, ако је $T < 0$ или $T \mathrel{!}= 0$ али је X празан скуп, можемо одмах рећи да не постоји тражени подскуп.

У општем случају, посматрајмо произвољан елемент x из скупа X(већ смо узели у обзир случај када је X празан скуп). Тада постоји подскуп од X чија је сума елемената једнака T када важи једна од следећих тврдњи:

- Постоји подскуп од X који укључује елемент x и чија је сума једнака T
- Постоји подскуп од X који не укључује елемент x и чија је сума једнака T

У првом случају мора постојати подскуп скупа $X\backslash\{x\}$ чија је сума елемената једнака $T-x$. У другом случају мора постојати подскуп скупа $X\backslash\{x\}$ чија је сума елемената једнака T. Стога, проблем суме подскупа можемо свести на два простија проблема. Резултујући алгоритам је приказан у псеудокоду 3.

```
<<Da li neki podskup od X ima sumu elemenata jednaku T?>>
SumaPodskupa(X, T):
        ako T = 0
                vrati Tačno
        inače ako T < 0 ili X je prazan skup
                vrati Netačno
        inače
                x = neki element iz skupa X
                sa = SumaPodskupa(X \ {x} ,T – x)   <<Rekurzija>>
                bez = SumaPodskupa(X \ {x} ,T)      <<Rekurzija>>
                vrati (sa ili bez)
```

Псеудокод 3 – Алгоритам који решава проблем сума подскупа

Коректност

Да је алгоритам коректан, једноставно се доказује индукцијом. Ако је *T = 0*, онда је збир елемената празног подскупа једнак *T*, тако да је *Тачно* коректан излаз. Иначе, ако *T* има негативну вредност или је *X* празан, ниједан подскуп од *X* нема збир *T*, па је *Netačno* коректан излаз. Иначе, ако постоји подскуп чији је збир елемената једнак *T*, подскуп садржи или не садржи *X[n]* и рекурзија проверава сваку од тих могућности.

Анализа

Да бисмо извршили анализу временске сложености алгоритма, треба да опишемо неколико имплементационих детаља прецизније. За почетак, претпоставимо да је скуп *X* дат као низ *X[1..n]*.

Претходни рекурзивни алгоритам нам омогућава да изаберемо било који елемент *x* из *X* у главном рекурзивном случају. Чисто због ефикасности, пожељно је изабрати такво *x* тако да преостали подскуп *X\{x}* има концизан опис (да буде једноставно и јасно репрезентован), који може брзо да се израчуна, како би рекурзиван позив имао минималне трошкове. Тачније, изабраћемо *x* који је последњи елемент *X[n]*. Тада је подскуп *X\{x}* сачуван у префиксу *X[1..n-1]*. Прослеђивање читаве копије овог префиска би трајало дуго, треба нам *θ(n)* времена за то, зато прослеђујемо само две вредности: референцу на низ и дужину префикса. Алтернативно, можемо избећи прослеђивање референце на *X* при сваком рекурзивном позиву, тако што дефинишемо *X* као глобалну променљиву. На овај начин добијамо алгоритам приказан псеудокодом 4.

```
<<Da li neki podskup od X[1..i] ima sumu elemenata jednaku T?>>
SumaPodskupa(X, i, T):
        ako T = 0
                vrati Tačno
        inače ako T < 0 ili i = 0
                vrati Netačno
        inače
                sa = SumaPodskupa(X, i - 1 ,T – X[i])      <<Rekurzija>>
                bez = SumaPodskupa(X, i - 1 ,T)            <<Rekurzija>>
                vrati (sa ili bez)
```

Псеудокод 4 – Унапређен алгоритам за проблем суме подскупа

Са оваквим избором имплементације, временска сложеност нашег алгоритма $T(n)$ задовољава рекурентну неједначину $T(n) \leq 2T(n-1) + O(1)$. Решење је $T(n) = O(2^n)$. У најгорем случају, нпр. када је T веће од суме свих елемената, стабло рекурзивних позива за овај алгоритам је комплетно бинарно стабло дубине n и алгоритам разматра свих 2^n подскупова од X.

Варијанте

Са малим изменама, можемо решити више варијанти проблема сума подскупа. Нпр. у псеудокоду 5 приказан је алгоритам који прави подниз од X, чија је сума свих елемената једнака T. Уколико такав подниз не постоји, функција враћа непостојећу вредност. Овај алгоритам користи исту рекурзивну стратегију као прошли алгоритам. Временска сложеност је такође $O(2^n)$. Анализа је најпростија ако претпоставимо да је временска сложеност убацивања елемента у низ $O(1)$, нпр. у листу. Временска сложеност алгоритма је $O(2^n)$, чак иако убацивање захтева $O(n)$ времена, нпр. у сортирану листу. Сличне варијанте омогућавају нам да пребројимо поднизове чија је сума свих елемената T или да изаберемо најбољи такав подниз (по неком критеријуму).

Већина других проблема који се могу решити бектрекингом имају следећу особину: иста рекурзивна стратегија може се искористити за различите варијанте истог проблема. Нпр. рекурзивна стратегија претходног проблема се лако може модификовати тако да враћа листу добрих потеза уместо тврдње да ли је тренутно стање игре добро или лоше. Стога, када пишемо бектрекинг алгоритам, потребно је тежити ка најпростијој могућој варијанти проблема, рачунајући број или једну логичку вредност уместо неких комплекснијих информација и структура.

```
<<Vraća podskup od X[1..i], tako da mu je suma elemenata jednaka T>>
<<Ili ako vrati nepostojeću vrednost, takav podskup ne postoji>>
KonstruišiPodksup(X, i, T):
    ako T = 0
        vrati ∅

    ako T < 0 ili n = 0
        vrati Nepostojeća_vrednost

    Y = KonstruišiPodksup(X, i - 1, T)
    ako Y ≠ Nepostojeća_vrednost
        vrati Y

    Y = KonstruišiPodksup(X, i - 1, T – X[i])
    ako Y ≠ Nepostojeća_vrednost
        vrati Y ∪ {X[i]}

    vrati Nepostojeća_vrednost
```

Псеудокод 5 – Варијанта проблема сума подскупа, која враћа један подскуп као решење

1.4 Општи образац

Бектрекинг алгоритми се обично користе за доношење *низа одлука*, са циљем изградње рекурзивно дефинисане структуре која задовољава одређена ограничења. Често, али не увек, ова циљна структура је сама по себи низ. На пример:

- Код проблема *n* дама, циљ је конструисати низ позиција дама, по једну у сваком реду, тако да се никоје две међусобно не нападају. За сваки ред, алгоритам *одлучује* где ће да постави даму.
- Код проблема стабла игре, циљ је конструисати низ потеза, тако да је сваки потез најбољи за играча који га повлачи. За свако стање игре, алгоритам *одлучује* који је најбољи могући следећи потез.
- Код проблема сума подниза, циљ је конструисати низ од унетих елемената чија је сума задат број. За сваки унет елемент, алгоритам *одлучује* да ли да га укључи у излазни низ или не.

У сваком рекурзивном позиву бектрекинг алгоритма, морамо направити тачно једну одлуку и наша одлука мора бити конзистентна са свим претходним одлукама. Стога, наш рекурзивни позив захтева не само део података које још нисмо обрадили, већ захтева и одговарајући резиме о одлукама које смо већ донели. Због ефикасности, резиме прошлих одлука би требао бити најмањи могући. Нпр:

- За проблем *n* дама, треба да проследимо не само број празних редова, већ и позиције претходно постављених дама. Овде, нажалост, морамо памтити све детаље прошлих одлука.
- У проблему стабла игре, треба да проследимо само тренутно стање игре и идентитет играча који је на потезу. Не морамо памтити ништа што се тиче прошлих одлука, јер ко ће победити из датог стања не зависи од потеза који су довели игру у то стање.
- За проблем суме подниза, морамо проследити преостале расположиве бројеве, као и преостали циљни број, који је разлика почетног циљаног броја и елемената који су већ одабрани. Није битно који су тачно елементи одабрани.

Када дизајнирамо нови бектрекинг алгоритам, морамо *унапред* знати које информације ће нам бити потребне у *сред алгоритма* о прошлим одлукама. Уколико ова информација није тривијална, наш рекурзивни алгоритам ће можда требати да реши општији проблем од оног који смо на почетку желели да решимо.

На крају, када схватимо који рекурзиван алгоритам заиста треба да решимо, решавамо тај проблем рекурзивном грубом силом. Испробавамо све могуће наредне одлуке, тако да су конзистентне са претходним одлукама и препустимо рекурзији да се брине о свему осталом. Дакле, не одсецамо очигледно лоше одлуке, већ их све испитујемо, а после можемо оптимизовати алгоритам.

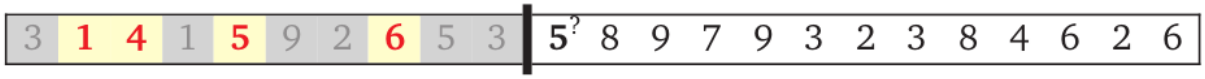
2 Најдужи растући подниз

Проблем гласи: за дати низ *S*, пронаћи **најдужи растући подниз**, односно најдужи подниз такав да су његови елементи у растућем поретку. За било који низ *S*, подниз од *S* је низ добијен од *S* брисањем нула или више елемената, без промене редоследа преосталих елемената. Елементи подниза не морају бити суседни у *S*. Нпр. када се возите великом градском улицом, пролазите кроз низ раскрсница са семафорима, али се морате зауставити само на поднизу тих раскрсница, на којима је црвено светло. Ако имате среће, нећете уопште стајати: празан низ је подниз од *S*. Са друге стране, ако баш немате среће, мораћете стати на свакој раскрсници: *S* је подниз самога себе.

Такође, ниске BENT, ACKACK, SQUARING и SUBSEQUENT су поднизови ниске SUBSEQUENCEBACKTRACKING, као што су и празна ниска и читава ниска SUBSEQUENCEBACKTRACKING, али ниске QUEUE, EQUUS и TALLYHO то нису. Подниз чији су елементи суседни у оригиналном низу се зову сегменти. Нпр. MASHER и LAUGHTER су поднизови MANSLAUGHTER, али само LAUGHTER је сегмент.

Претпоставимо сада да је дат низ целих бројева и треба да пронађемо најдужи подниз чији су елементи у растућем поретку. Конкретније, улаз је низ целих бројева *A* дужине *n*, и треба да израчунамо најдужи могући низ индекса $1 \le i_1 < i_2 < \ldots < i_l \le n$ тако да је $A[i_k] < A[i_{k+1}]$ за свако k.

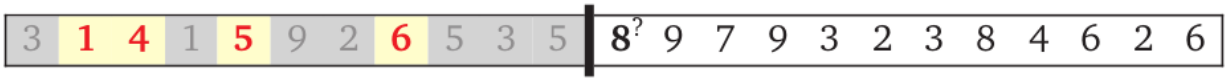
Један природни приступ проналажења најдужег растућег подниза је одлучивати, за сваки индекс j који иде од 1 до *n*, да ли укључити *A[j]* у подниз. Проналазећи се у сред низа одлучивања, можемо замислити стање на слици 5.



Слика 5 – Стање претраге за пример који се дискутује

Црна преграда раздваја део улаза који смо обрадили од дела улаза који још нисмо обрадили. Бројеви које смо већ одлучили да укључимо у тражени подниз су обојени у црвено, док су бројеви које смо одлучили да искључимо обојени у сиво. Приметимо да су бројеви које смо одлучили да укључимо у тражени подниз у растућем поретку. Наш алгоритам мора одлучити да ли ће да укључи или не број који се налази иза црне преграде.

У овом примеру дефинитивно не можемо укључити број 5, јер онда изабрани бројеви не би били у растућем поретку. Следеће стање претраге је приказано на слици 6.



Слика 6 – Стање претраге за пример који се дискутује

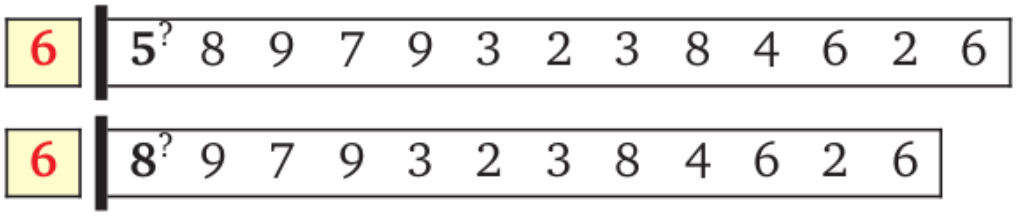
Сада можемо укључити 8, али није очигледно да ли бисмо то требали да урадимо. Радије него да одсецамо лоше одлуке, наш бектрекинг алгоритам ће једноставно користити грубу силу.

- Прво пробамо да укључимо 8 и препустимо рекурзији остале одлуке
- Затим пробамо да искључимо 8 и препустимо рекурзији остале одлуке

Која год од ових одлука води дужем растућем поднизу је права одлука.

Сада је кључно питање: шта требамо памтити о прошлим одлукама? Можемо укључити *A[j]* само ако је резултујући подниз растући. Ако претпоставимо(индуктивно!) да су претходно одабрани бројеви из *A[1..j-1]* у растућем поретку, онда можемо укључити *A[j]* ако и само ако је *A[j]* већи

од последњег одабраног броја из *A[1..j-1]*. Стога, треба нам само информација о последњем одабраном броју. Сада можемо преправити слику 6, бришући све што нам не треба и добијамо слику 7.



Слика 7 – Стање претраге за пример који се дискутује са избаченим непотребним подацима

Проблем који наша рекурзивна стратегија решава је следећи:

> Ако је задат претходни број *prethodni* и низ A[1..n], пронаћи најдужи растући подниз од A у којем је сваки елемент већи од *prethodni*.

Као и обично, наша рекурзивна стратегија захтева базни случај. Наша тренутна стратегија се зауставља када дође до краја низа, јер не постоји следећи број за разматрање. Празан низ има тачно један подниз и то је празан низ. Можемо рећи да је сваки елемент у празном низу већи од било које вредности и сваки пар елемената у празном низу је у растућем поретку. Стога, најдужи растући подниз празног низа има дужину 0.

Резултујући рекурзивни алгоритам је приказан у псеудокоду 6.

```
NRPveći(prethodni, A[1..n]):
    ako n = 0
        vrati 0
    inače ako A[1] ≤ prethodni
        vrati NRPveći(prehodni, A[2..n])
    inače
        preskoči = NRPveći(prethodni, A[2..n])
        uključi = NRPveći(A[1], A[2..n]) + 1
        vrati max{preskoči, uključi}
```

Псеудокод 6 – Рекурзивни алгоритам за проблем најдужи растући подниз

Сетимо се да је прослеђивање низова скупоцено за системски стек. Пробајмо преформулисати све у терминима индекса низа, претпостављајући да је *A[1..n]* глобална променљива. Број *prethodni* је типично елемент низа *A[i]* и остатак низа је увек суфикс *A[j..n]* оригиналног улазног низа. Сада можемо преформулисати наш рекурзивни проблем на следећи начин:

> Ако су дата два индекса *i* и *j*, где је *i* < *j*, пронаћи најдужи растући подниз од *A[j..n]* у коме је сваки елемент већи од *A[i]*.

Нека *NRPveći*(i,j) означава најдужи растући подниз од *A[j..n]* у којем је сваки елемент већи од *A[i]*. Наша рекурзивна стратегија даје следећу рекурзивну дефиницију, приказану на слици 8.

$$NRPveći(i,j) = \begin{cases} 0, & \text{ako je } j > n \\ NRPveći(i, j+1) & \text{ako je } A[i] \geq A[j] \\ \max\{NRPveći(i,j+1), 1 + NRPveći(j,j+1)\}, & \text{inače} \end{cases}$$

Слика 8 – Рекурзивна дефиниција функције *NRPveći*

Алтернативно, можемо да је запишемо у форми псеудокода приказаног у псеудокоду 7.

```
NRPveći(i, j):
        ako i > n
                vrati 0
        inače ako A[i] ≥ A[j]
                vrati NRPveći(i, j + 1)
        inače
                preskoči = NRPveći(i, j + 1)
                uključi = NRPveći(j, j + 1) + 1
                vrati max{preskoči, uključi}
```

Псеудокод 7 – Функција *NRPveći* у форми псеудокода

Коначно, треба да повежемо нашу стратегију са оригиналним проблемом: „Пронађи најдужи растући подниз низа без других услова. Најједноставнији приступ би био да додамо вештачки елемент -∞ на почетак низа (Псеудокод 8).

```
NRP(A[1..n]):
        A[0] = -∞
        vrati NRPveći(0,1)
```

Псеудокод 8 – Функција NRP која решава почетни проблем

Временска сложеност функције *NRPveći*, иста је као и рекурзивне функције која решава проблем куле Ханоја, тј. $T(n) \leq 2T(n-1) + O(1)$, што имплицира да је $T(n) = O(2^n)$. Не треба да нас изненади ова временска сложеност. У најгорем случају, алгоритам разматра свих 2^n поднизова улазног низа.

3 Задаци

3.1 *N* дама

За шаховску таблу димензија ***n*** *x* ***n***, одредити на колико начина је могуће на њу поставити ***n*** дама, тако да се никоје две међусобно не нападају.

Улаз: Са стандардног улаза се величина шаховске табле **n**.

Излаз: На стандардни излаз исписати број различитих решења за постављање ***n*** дама на шаховску таблу димезија ***n*** *x* ***n***.

Пример:

-Улаз: *8*
-Излаз: *Broj resenja problema n dama za tablu dimenzija 8 x 8 je: 92*

-Улаз: *10*
-Излаз: *Broj resenja problema n dama za tablu dimenzija 10 x 10 je: 724*

Код:

```
#include<iostream>
#include<vector>

using namespace std;

int n;

int ndama(vector<vector<bool>> &tabla, int r){
	if (r == n)
		return 1;
	int broj_resenja=0;
	for(int i=0; i<n; i++){ //razmatra se postavljanje dame sva polja u redu r
		bool legalno_postavljanje = true;
		//proverava se da li je postavljanje da me na i-to polje
		//u konfliktu sa prethodno postavljenim damama u prvih r-1 redova
		for(int j=0; j<r; j++){
			if (tabla[j][i] || (i-r+j>=0 && tabla[j][i-r+j])
					|| (i+r-j<n && tabla[j][i+r-j])){
				legalno_postavljanje = false;
				break;
		}
		}
		//ako nije u konfliktu, postavi damu na i-to polje i
		//rekurzivno pozovi funkciju za postavljanje dama na preostale prazne redove
		if(legalno_postavljanje){
			tabla[r][i]=true;
			broj_resenja+= ndama(tabla, r+1);
			tabla[r][i]=false;
```

```
            }
        }
        return broj_resenja;
}

int main(){
        cout << "Unesite velicinu sahovske table" << endl;
        cin >> n;
        //false oznacava da se dama ne nalazi na polju, a true da se ne nalazi
        vector<vector<bool>> tabla(n, vector<bool>(n, false));

        int broj_resenja = ndama(tabla,0);
        cout << "Broj resenja problema n dama za tablu dimenzija " << n << " x "
                << n  << " je: " << broj_resenja << endl;
}
```

Решење

Са стандардног улаза се уноси величина шаховске табле ***n*** која се чува у глобалној променљивој. Рекурзивној функцији се на почетку прослеђује празна табла репрезентована као матрица булових вредности, где вредност нетачно означава да на поље није постављена дама. Такође се прослеђује и параметар ***r*** који означава тренутан ред у који постављамо даму. Постављамо даме ред по ред, почевши од првог реда и на почетку је ***r*** = 0. Разматра се постављање даме на свако поље у реду ***r***, и уколико постављање није у конфликту са претходно постављеним дама (у редовима од 0-тог до (***r***-1)-ог), дама се поставља на то поље и рекурзивно се позива функција за постављање дама у преостале празне редове.

Уколико је ***r*** = ***n***, то значи да смо успешно поставили **n** dama, и функција враћа 1 означавајући да је пронађено једно решење. Ако у неком не успемо да поставимо иједну даму функција враћа 0. Уводи се бројач који чува збир свих повратних вредности рекурзивних позива као коначно решење за овај проблем.

3.2 Сума подскупа

Описати рекурзиван алгоритам за следећу генерализацију проблема сума подскупа:

(а) Дат је низ природних бројева ***X[1..n]*** и цели број ***T***. Израчунати број подскупова чија је сума елемената једнака ***T***.

(б) Дата су два низа природних бројева ***X[1..n]*** и ***W[1..n]*** и цели број ***T***, где сваки **W[i]** означава тежину одговарајућег елемента ***X[i]***. Израчунати тежину подскупа од ***X*** највеће тежине, чија је сума елемената једнака ***T***. Ако не постоји такав подскуп, алгоритам треба да врати *„-inf“*.

(а)

Улаз: Са стандардног улаза уноси се скуп природних бројева и тражена сума подскупа ***T***. Уноси се *0* за крај скупа.

Излаз: На излаз се исписује број подскупова чија је сума елемената једнака ***T***.

Пример:

-Улаз: $X=\{8,6,7,5,3,10,9\}$ и $T=15$

-Излаз: 4 (то су подскупови $\{8,7\}$, $\{7,5,3\}$, $\{6,9\}$ и $\{5,10\}$)

Код:

```
#include<iostream>
#include<vector>
using namespace std;

int subsetsNumber(vector<int>& skup, int duzina, int T){
	// pronasli smo jedan podskup ciji elementi imaju trazenu sumu
	if (T==0) return 1;
	// sa ovim elementima ne moze da se izgradi ciljni podskup
	// jer prevazilazi trazenu sumu
	// ili nemamo vise elemenata za razmatranje
	else if (T<0 || duzina==0) return 0;
	else{
		// inace pokusavamo sa ubacivanjem trenutnog elementa
		// koji se razmatra
		// ili sa njegovim preskakanjem
		int with = subsetsNumber(skup, duzina-1, T-skup[duzina-1]);
		int wout = subsetsNumber(skup, duzina-1, T);
		return with + wout;
	}
}

int main(){
	vector<int> skup;
	int x,T;

	cout << "Unesite elemente skupa i 0 za kraj unosa" << endl;
	while (1){
		cin >> x;
		if (x==0) break;
		skup.push_back(x);
	}

	cout << "Unesite broj T" << endl;
	cin >> T;
	int brojPodskupova = subsetsNumber(skup,skup.size(),T);
	cout << brojPodskupova << endl;
}
```

Решење

Рекурзивно се обилазе сви подскупови у дубину и броји се број подскупова чија је сума елемената ***T***. Функцији се прослеђује скуп бројева као низ. Она покушава да креира тражени збир са и без последњег елемента у низу који се тренутно разматра. Функцији се прослеђује и дужина префиска низа који се разматра за бирање елемената, као и број ***T*** који представља преосталу суму коју треба да креирамо од преосталих елемената низа (скупа). Ако за пронађени скуп важи да је сума његових елемената

једнака ***T***, функција враћа 1, иначе враћа 0. Број подскупова чија је сума елемената једнака ***T*** је сума свих повратних вредности.

(б)

Улаз: Са стандардног улаза уноси се скуп природних бројева, тежине његових елемената и тражена сума подскупа ***T***. Уноси се *0* за крај скупа.

Излаз: На излаз се исписује тежина најтежег подскупа чија је сума елемената једнака ***T*** или ако не постоји ни један скуп чија је сума елемената једнака ***T*** исписати *„-inf“*.

Пример:

-Улаз: *X={8,6,7,5,3,10,9}, W={1,2,3,4,5,6,7}* и *T=15*

-Излаз: *12 (најтежи подскуп је {7,5,3})*

Код:

```
#include<iostream>
#include<vector>
#include<bits/stdc++.h>

using namespace std;

int subsetsNumber(vector<int> &skup, vector<int> &tezine, int duzina, int T,
                  int ukupnaTezina){
        // pronadjen je jedan podskup ciji elementi imaju trazenu sumu
        // i vracamo njegovu tezinu
        if( T==0) return ukupnaTezina;
        else if (T<0 || duzina==0) return INT_MIN;
        else{
                int with = subsetsNumber(skup, tezine, duzina-1, T-skup[duzina-1],
                                         ukupnaTezina+tezine[duzina-1]);
                int wout = subsetsNumber(skup, tezine, duzina-1, T, ukupnaTezina);
                // biramo skup sa vecom tezinom
                return with > wout ? with : wout;
        }
}

int main(){
        vector<int> skup,tezine;
        int x,T;

        cout << "Unesite elemente skupa i 0 za kraj unosa" << endl;
        while (1){
                cin >> x;
                if (x==0)
                        break;
                skup.push_back(x);
        }

        cout << "Unesite tezine skupa i 0 za kraj unosa" << endl;
        while (1){
```

```
            cin >> x;
            if (x==0)
                  break;
            tezine.push_back(x);
      }

      cout << "Unesite broj T" << endl;
      cin >> T;
      int tezinaNajtezegPodskupa = subsetsNumber(skup,tezine,skup.size(),T,0);

      if (tezinaNajtezegPodskupa==INT_MIN)
            cout << "-inf" << endl;
      else
            cout << tezinaNajtezegPodskupa << endl;
}
```

Решење

Функција такође рекурзивно обилази све подскупове у дубину, као и у примеру под *а)*. Њој се додатно прослеђују низ тежина елемената и тренутна укупна тежина одабраних елемената. Повратна вредност је тежина најтежег подскупа са траженим збиром елемената уколико он постоји, иначе је минус бесконачно.

3.3 Адитивни ланац

Адитивни ланац за природан број ***n*** је растући низ природних бројева који почиње са 1 и завршава се са ***n***, тако да је сваки његов елемент осим првог збир нека два елемента која му претходе. Формалније речено, низ природних бројева $x_0 < x_1 < x_2 < \ldots < x_l$ је адитивни ланац за ***n*** ако и само ако важи:

- $x_0 = 1$
- $x_l = n$, и
- за сваки индекс k ($l \geq k > 0$), постоје индекси $0 \leq i \leq j < k$, тако да је $x_k = x_i + x_j$

Дужина адитивног ланца је број елемената минус 1 (не бројимо први елемент). Нпр. [1,2,3,5,10,20,23,46,92,184,187,374] је адитивни ланац броја 374 дужине 11.

Написати програм који проналази адитивни ланац минималне дужине за дат природан број ***n***.

Улаз: Са стаднардног улаза се уноси природан број *n*.

Излаз: На стандардни излаз исписати адитивни ланац минималне дужине за дато n.

Пример:

```
-Улаз:     374
-Излаз:    Duzina aditivnog lanca minimalne duzine je 11
           Jedan takav lanac je:
           1 2 4 8 16 32 64 80 82 146 292 374
```

-Улаз: *533*
-Излаз: *Duzina aditivnog lanca minimalne duzine je 12*
Jedan takav lanac je:
1 2 4 8 16 32 64 128 256 512 528 532 533

Код:

```
#include<iostream>
#include<vector>
#include<math.h>
using namespace std;

vector<int> trenutni_optimalni_niz;
int trenutna_optimalna_duzina;
int n;

void kopiraj_niz(vector<int> &niz1, vector<int> &niz2){
	niz1.resize(niz2.size());
	for(int i=0; i<niz2.size(); i++){
		niz1[i]=niz2[i];
	}
}

void ispisi_rez(){
	cout << "Duzina aditivnog lanca minimalne duzine je " <<
	trenutna_optimalna_duzina << endl;
	cout << "Jedan takav lanac je:  ";
	for(int i=0; i<trenutni_optimalni_niz.size(); i++)
		cout << trenutni_optimalni_niz[i] << " ";
	cout << endl;
}

void aditivni_niz(vector<int> &niz, int duzina_niza){
	// azuriramo vrednost novog najkraceg do sada pronadjenog
	// aditivnog lanca
	if (niz[duzina_niza]==n){
		kopiraj_niz(trenutni_optimalni_niz, niz);
		trenutna_optimalna_duzina=duzina_niza;
		return;
	}
	for(int i=duzina_niza; i>=0; i--){
		for(int j=i; j>=0; j--){
			int sledeci_element = niz[i]+niz[j];
			if (sledeci_element > niz.back() && sledeci_element <= n &&
			duzina_niza+1<trenutna_optimalna_duzina){
				niz.push_back(sledeci_element);
				aditivni_niz(niz, duzina_niza+1);
				niz.pop_back();
			}
		}
	}
}
```

```
int main(){
	cin >> n;

	vector<int> niz;
	niz.push_back(1);

	trenutna_optimalna_duzina=2*ceil(log2(n)+1)+1;

	aditivni_niz(niz,0);
	ispisi_rez();
}
```

Решење

Адитивни ланац се гради елемент по елемент, кренувши од једночланог низа са елементом 1. Рекурзивној функцији се прослеђује до сада изграђен низ. Уколико је последњи елемент једнак ***n***, то значи да је успешно креиран један адитивни ланац и проверава се да ли је он најмање дужине од свих који су до сада пронађени тако што се пореди са тренутно накраћим ланцем који се чува у глобалној променљивој. Иначе се покушава додавање новог елемента који је збир нека два од претходних елемената и функција врши рекурзиван позив за нови ланац који је дужи за један елемент. Врши се одсецање и нови елемент се не додаје уколико је он већи од ***n*** или уколико се његовим додавањем превазилази дужина до сада најкраћег пронађеног адитивног ланца. Додатно, можемо да приметимо да је најкраћи адитивни ланац сигурно краћи од $2*([log_2(n)]+1)+1$ и због тога од почетка нећемо креирати низове који су дужи од овог броја. То важи зато што број **n** можемо представити као збир $[log_2(n)]+1$ различитих бројева који су степени двојке, а те бројеве инкрементално добијамо сабирајући последњи елемент са самим собом док градимо низ. Онда нам још треба највише $[log_2(n)]+1$ сабирања да бисмо добили ***n***.

3.4 Најдужи заједнички подниз

Нека су ***A[1..m]*** и ***B[1..n]*** два произвољна низа карактера. Заједнички подниз од ***A*** и ***B*** је низ који је подниз и од ***A*** и од ***B***. Написати рекурзивну функцију *lcs(**A,B**)*, која враћа дужину најдужег заједничког подниза од ***A*** и ***B***.

Улаз: Са стандардног улаза се уносе два низа карактера ***A*** и ***B***.

Излаз: На стандардни излаз исписати дужину најдужег заједничког подниза ***A*** и ***B***.

Пример:

```
-Улаз:     ABCBX
           ABDCAB
-Излаз:    Najduzi zajednicki podniz niski a i b je duzine 4
           Jedan takav podniz je: ABCB

-Улаз:     XMJYAUZ
           MZJAWXU
-Излаз:    Najduzi zajednicki podniz niski a i b je duzine 4
           Jedan takav podniz je: MJAU
```

Код:

```
#include<iostream>
#include<string>
using namespace std;

int max_duzina=0;
string max_string="";

void lcs(string a, int pa, string b, int pb, string tekuci, int duzina){
	// pronasli smo jedan dvostruki podniz i proveravamo
	// da li je duzi od trenutno najduzeg pronadjenog
	// u tom slucaju azuriramo vrednost max_string
	if (pa==a.size() || pb==b.size()){
		if (duzina>max_duzina){
			max_duzina = duzina;
			max_string = tekuci;
		}
		return;
	}
	for(int i=pa; i<a.size(); i++){
		// pokusavamo da izgradimo dvostruki podniz
		// bez karaktera koji se nalazi na poziciji i u stringu a
		// i rekurzivno obradjujemo ostatak niski
		lcs(a,i+1,b,pb,tekuci,duzina);

		// pokusavamo da izgradimo dvostruki podniz
		// sa karakterom koji se nalazi na poziciji i u stringu a
		// i trazimo isti takav karakter u niski b
		// ukoliko ga pronadjemo rekurzivno obradjujemo ostatak niski
		// trenutna duzina se uvecava za 1
		int j=pb;
		while (j<b.size() && b[j]!=a[i]){
			j++;
		}
		if (j<b.size()){
			tekuci.push_back(a[i]);
			lcs(a,i+1,b,j+1,tekuci,duzina+1);
			tekuci.pop_back();
		}
	}
}

void lcs(string a, string b){
	string tekuci;
	lcs(a,0,b,0,tekuci,0);
}

int main(){
	string a,b;
	cout << "Unesite string a: ";
```

```
    cin >> a;
    cout << "Unesite string b: ";
    cin >> b;

    lcs(a,b);
    cout << "Najduzi zajednicki podniz niski a i b je duzine " << max_duzina << endl;
    cout << "Jedan takav podniz je: " << max_string << endl;
}
```

Решење

Најдужи заједнички подниз се проналази тако што се претражују сви заједнички поднизови и бира се најдужи. Рекурзивној функцији се прослеђују ниске ***a*** и ***b***, индекси ***pa*** и ***pb*** које представљају почетне позиције подниски ***a*** и ***b*** које још нисмо обрадили, као и текући заједнички подниз ***tekuci*** који се гради и његова дужина ***duzina***. Крећемо са обрађивањем ниски од њиховог почетка.

Стабло рекурзивних позива се гради тако што пролазимо кроз карактере ниске ***a*** и обрађују се случајеви када карактер на позицији ***pa*** додајемо у ниску ***tekuci***, такође и када га не додајемо. У случају када се карактер на позицији ***pa*** не додаје у текући, рекурзивном позиву само се увећавава ***pa*** за 1, а остали аргументи остају исти. У другом случају у којем се тај карактер додаје текућем, тражи се прва појава тог карактера у необрађеном делу ниске ***b***. Ако се он пронађе у ниски ***b***, позива се рекурзија са додатим карактером у ниску ***tekuci*** и помереним индексима ***pa*** и ***pb*** десно од карактера које смо упарили. На овај начин обилазимо све могуће кандидате за најдужи заједнички подниз и у току претраге ажурирамо вредност тренутног највећег подниза који се чува у глобалној променљивој.

3.5 Двоструки подниз

Дата су два низа карактера **X[1..k]** и **Y[1..n]**, тако да је **k≤n**. Написати рекурзиван бектрекинг алгоритам који одређује да ли је **X** двоструки подниз од **Y**, тј. да ли се **X** појављује у **Y** као два дисјунктна подниза. Нпр. PPAP је двоструки подниз од PENPINEAPPLEAPPLEPEN (два дисјунктна подниза су обојена жутом и плавом бојом).

Улаз: Са стандрадног улаза се уносе два низа карактера **X** и **Y**.

Излаз: На стандардни излаз исписати одговор на питање: „Да ли је **X** двоструки подниз од **Y**?“

Пример:

```
-Улаз:     PALE
           PENPINEAPPLEAPPLEPEN
-Излаз:    Niska X jeste dvostruki podniz niske Y
           Pozicije prve i druge niske su:
           10020001001120022000
           (PENPINEAPPLEAPPLEPEN)


-Улаз:     PLEN
           PENPINEAPPLEAPPLEPEN
-Излаз:    Niska X nije dvostruki podniz niske Y
```

Код:

```
#include<iostream>
#include<vector>
#include<string>
using namespace std;

string x,y;

bool dvostruki_podniz(int py, int p1, int p2, vector<int> &oznaka){
	// izgradili smo dvostruki podniz
	if (p1 == x.size() && p2 == x.size()) return true;
	// dosli smo do kraja niske Y
	// i nismo izgradili dvostruki podniz
	if (py==y.size()) return false;
	// nadogradjujemo prvi podniz ako se sledeci trazeni karakter
	// poklapa sa karakterom niske Y na poziciji py
	// i rekurzivno obradjujemo ostatak niske Y
	if (p1<x.size() && y[py]==x[p1]){
		oznaka[py]=1;
		if (dvostruki_podniz(py+1, p1+1, p2, oznaka)) return true;
		oznaka[py]=0;
	}
	// nadogradjujemo drugi podniz ako se sledeci trazeni karakter
	// poklapa sa karakterom niske Y na poziciji py
	// i rekurzivno obradjujemo ostatak niske Y
	if (p2<x.size() && y[py]==x[p2]){
		oznaka[py]=2;
		if (dvostruki_podniz(py+1, p1, p2+1, oznaka)) return true;
		oznaka[py]=0;
	}
	// preskacemo karakter niske Y na poziciji py
	// i rekurzivno obradjujemo ostatak niske Y
	if (dvostruki_podniz(py+1,p1,p2,oznaka)) return true;
	return false;
}

void ispisi(vector<int> oznaka){
	for(int i=0; i<oznaka.size(); i++){
		cout << oznaka[i] << " ";
	}
	cout << endl;
}

int main(){
	cout << "Unesite nisku X: " << endl;
	cin >> x;
	cout << "Unesite nisku Y: " << endl;
	cin >> y;

	vector<int> oznaka(y.size(), 0);
```

```
	if (dvostruki_podniz(0,0,0,oznaka)){
		cout << "Niska X jeste dvostruki podniz niske Y" << endl;
		cout << "Pozicije prve i druge niske su: " << endl;
		ispisi(oznaka);
	}
	else{
		cout << "Niska X nije dvostruki podniz niske Y" << endl;
	}
}
```

Решење

Ниске ***X*** и ***Y*** се уносе са стандардног улаза и чувају се у глобалној променљивој. Ниска ***Y*** се обрађује карактер по карактер са лева на десно и траже се карактери који се налазе у ниски ***X***. Циљ је пронаћи два пута подниз од ***Y*** који је једнак ниски ***X***, тако да се њихови карактери налазе на различитим позицијама у оквиру ниске ***Y***.

Рекурзивној функцији се прослеђују ***py***, тј. позиција текућег карактера ниске ***Y*** који обрађујемо, као и бројеви ***p1*** и ***p2*** који означавају са колико смо карактера до сада изградили прву и другу тражену ниску. Додатно прослеђујемо и помоћни низ са којим означавамо које смо карактере из ***Y*** доделили првој и другој ниски. Ако карактер из ***Y*** на позицији ***py*** можемо да надоградимо на тренутно изграђену прву ниску, то радимо тако шпо увећавамо бројеве ***py*** и ***p1***, затим то означавамо у низу ***oznaka*** и рекурзивно позивамо функцију да обради остатак ниске ***Y***. Аналогоно радимо то проверавајући да ли можемо са тим карактером да надоградимо и другу тражену ниску. Такође покушавамо да направимо тражене подниске и без карактера ниске ***Y*** на позицији ***py***, тако што само увећавамо индекс ***py*** и вршимо рекурзиван позив за остатак низа. Функција враћа *true* уколико је пронађено решење, иначе *false*.

3.6 Излаз из лавиринта

Наш пријатељ Пера се пробудио у лавиринту и не зна како је ту доспео. Поделио је локацију са нама преко *GPS*-а и пронашли смо лавиринт у којем се налази преко *Google Earth* апликације. Треба да помогнемо другу Пери да изађе из лавиринта, тако што ћемо му рећи којим путем да иде.

Улаз: Са стандардног улаза се уносе се бројеви ***n*** > 0 и ***m*** > 0, матрица целобројних вредности димензија ***n*** *x* ***m***, која представља лавиринт и координате друга Пере ***r*** и ***k*** у лавиринту. У матрици су са 1 означене препреке у лавиринту, а са 0 слободан простор којим Пера може да се креће.

Излаз: Пут којим Пера може да изађе из лавиринта.

Пример:

-Улаз:
```
12 13
1 0 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 0 1 0 1
1 0 0 0 1 0 0 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 0 0 0 0 1
1 0 1 1 1 0 1 1 1 1 1 0 1
1 0 0 0 0 0 0 0 0 0 0 0 1
1 0 1 1 1 0 1 1 1 0 1 1 1
1 0 0 0 1 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1
5 6
```

-Излаз:
```
Pronasli smo put kojim ce Pera izaci iz lavirinta!
Perin put pocinje od pozicije sa vrednoscu 2 i krece se
redom brojevima na vise.
1 28  1  1  1  1  1  1  1  1  1  1  1
1 27 26 25 24 23 22 21 20 19 18 17  1
1  1  1  1  1  1  1  1  1  0  1 16  1
1  0  0  0  1  0  0  0  1  0  1 15  1
1  0  1  0  1  0  1  0  1  0  1 14  1
1  0  1  0  1  0  1  0  1  0  1 13  1
1  0  1  0  1  2  1  0  0  0  0 12  1
1  0  1  1  1  3  1  1  1  1  1 11  1
1  0  0  0  0  4  5  6  7  8  9 10  1
1  0  1  1  1  0  1  1  1  0  1  1  1
1  0  0  0  1  0  0  0  0  0  0  0  1
1  1  1  1  1  1  1  1  1  1  1  1  1
```

-Улаз:
```
12 13
1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 0 1 0 1
1 0 0 0 1 0 0 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 0 0 0 0 1
1 0 1 1 1 0 1 1 1 1 1 0 1
1 0 0 0 0 0 0 0 0 0 0 0 1
1 0 1 1 1 0 1 1 1 0 1 1 1
1 0 0 0 1 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1
5 6
```

-Излаз:
```
Nismo pronasli put za Peru! Moracemo da posaljemo
helikopter po njega!
```

Код:

```
#include<iostream>
#include<vector>
#include<utility>
#include <iomanip>
using namespace std;
```

```
int n,m;
vector<pair<int,int>> kretanje = {{0,1}, {1,0}, {-1,0}, {0,-1}};

void ispisi(vector<vector<int>> &lavirint){
	for(int i=0; i<n; i++){
		for(int j=0; j<m; j++){
			cout << setw(2) << lavirint[i][j] << " ";
		}
		cout << endl;
	}
	cout << endl;
}

bool pronadji_put(vector<vector<int>> &lavirint,
			vector<vector<bool>>	&pretrazena_mesta, int korak, int r, int k){
	lavirint[r][k]=korak; // obelezavamo korake kojim trazimo izlaz iz lavirinta
	// obelezavamo mesta koja smo posetili, kako ih ne bismo ponovo pretrazivali
	// i kako ne bismo usli u ciklus
	pretrazena_mesta[r][k]=true;
	if (r==0 || k==0 || r==n-1 || k==m-1)
		return true;// Pronasli smo izlaz
	for(int i=0; i<kretanje.size(); i++){
		// razmatramo moguca kretanja iz trenutne pozicije
		int r1 = r + kretanje[i].first;
		int k1 = k + kretanje[i].second;
		if(lavirint[r1][k1]!=1 && !pretrazena_mesta[r1][k1]){
			if (pronadji_put(lavirint, pretrazena_mesta, korak+1, r1, k1))
				return true; // pronasli smo put
		}
	}
	// nismo nasli put kroz ovo polje, pa brisemo korake koje smo obelezili
	lavirint[r][k]=0;
	return false;
}

int main(){
	cout << "Unesite dimenzije lavirinta n i m" << endl;
	cin >> n >> m;
	cout << "Unesite lavirint u formi matrice "<< n << " x " << " m," << endl;
	cout << "1 za prepreke, a 0 za slobodan prostor" << endl;
	vector<vector<int>> lavirint(n, vector<int>(m));
	for(int i=0; i<n; i++)
		for(int j=0; j<m; j++)
			cin >> lavirint[i][j];
	int r,k;
	cout << "Unesite koordinate druga Pere u lavirintu" << endl;
	cin >> r >> k;

	vector<vector<bool>> pretrazena_mesta(n,vector<bool>(m,false));
	// brojimo korake pri izgradnji puta pocevski brojanje od 2
	// posto su 0 i 1 zauzeti i imaju drugo znacenje
```

```
        if(pronadji_put(lavirint,pretrazena_mesta,2,r,k)){
                cout << "Pronasli smo put kojim ce Pera izaci iz lavirinta!" << endl;
                cout << "Perin put pocinje od pozicije sa vrednoscu 2 i krece se redom
                        brojevima na vise." << endl;
                ispisi(lavirint);
        }
        else{
                cout << "Nismo pronasli put za Peru! Moracemo da posaljemo helikopter po
                        njega!" << endl;
        }

}
```

Решење:

Рекурзивно претражујемо путеве кроз лавиринт у дубину. Обележавамо кораке које правимо при претрази и памтимо места која смо посетили, како не бисмо ушли у циклус или претраживали више пута из места кроз које нисмо пронашли пут до излаза из лавиринта. Од Перине почетне позиције разматрамо креирање пута ка излазу у сва 4 смера (горе, доле, лево и десно) и рекурзивно позивамо даљу претрагу.

Кораке у матрици обележавамо бројевима почевши од броја 2, пошто су са 0 и 1 означени слободни простор и препреке.