# Конструкција и анализа алгоритама

Миодраг Живковић

## 1 Увод

Сврха овог излагања (видети уводни део књиге [1]) је увод у начин размишљања приликом конструкције и анализе алгоритама. Биће приказан начин специфицирања алгоритама, две основне стратегије за конструкцију алгоритама и неке основне идеје које се користе при анализи алгоритама, односно доказивању коректности алгоритама.

Два алгоритма који ће бити приказани решавају проблем сортирања (уређивања у неопадајућем поретку) датог низа од $n$ бројева. Алгоритми се излажу у облику псеудокода, који, иако се не може директно превести у било који стандардни програмски језик, описује структуру алгоритма довољно јасно, тако да програмер може да га реализује на програмском језику по свом избору. Приказују се алгоритам сортирања уметањем, који користи инкрементални приступ, и сортирање обједињавањем, алгоритам који користи идеју разлагања (divide-and-conquer) Иако време потребно за извршавање ова два алгоритма расте са $n$, брзине раста тих времена нису исте. Одредићемо времена извршавања ових алгоритама и упознати се са корисном нотацијом за њихово изражавање.

## 2 Алгоритми

Неформално, **алгоритам** је прецизно дефинисана процедура израчунавања, која, полазећи од неке вредности или скупа вредности као **улаза**, производи неку вредност, или скуп вредности, као **излаз**.

Алгоритам такође можемо посматрати као средство за решавање прецизно дефинисаног (рачунарског) **проблема**. Поставка проблема прецизира жељени однос излаза са улазом. Алгоритам описује специјалну процедуру израчунавања којом се постиже жељени однос излаза са улазом.

На пример, нека је потребно поређати чланове датог низа бројева тако да чине неопадајући низ. На овај проблем се често наилази; он може да послужи као добра подлога за увођење многих стандардних техника за конструкцију и анализу. Формално се проблем **сортирања** дефинише на следећи начин:

**Улаз:** Низ бројева $(a_1, a_2, \ldots, a_n)$.

**Излаз:** Пермутација $(a'_1, a'_2, \ldots, a'_n)$ улазног низа таква да је $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

На пример, за дати низ $(31, 41, 59, 26, 41, 58)$ алгоритам сортирања као излаз даје низ $(26, 31, 41, 41, 58, 59)$. Сваки овакав улазни низ зове се **инстанца** проблема сортирања. Уопште, **инстанцу проблема** чини улаз (који задовољава сва ограничења која проистичу из формулације проблема) неохподан да би се израчунало решење проблема.

Сортирање је фундаментална операција (многи програми је користе као међукорак), па је због тога развијен велики број добрих алгоритама за сортирање. Који алгоритам је најбољи за конкретну примену зависи, поред осталог, од дужине низа који треба сортирати, од могућих ограничења за чланове низа, од тога у којој су мери чланови низа већ сортирани, као и од типа уређаја у коме је низ смештен (оперативна меморија, диск или магнетска трака).

Каже се да је алгоритам **коректан** ако за се за сваку инстанцу завршава са тачним излазом. Тада кажемо да алгоритам **решава** дати проблем.

Алгоритам може бити специфициран на српском језику, као рачунарски програм, или чак као пројекат уређаја који га извршава. Једини захтев је да спецификација мора да обезбеди прецизан опис процедуре израчунавања.

Сортирање није једини проблем за чије решавање су развијени алгоритми. Наводимо примере проблема који се решавају алгоритмима:

- Један од циљева пручавања људског генома је идентификација свих око 100000 гена у људској ДНК, одређивање редоследа око три милијарде базних парова који чине људску ДНК, смештање тих информација у базе података, и развијање алата за анализу података. Сваки од тих корака захтева софистициране алгоритме. Циљ је постизање уштеда у утрошеном времену, и за људе и рачунаре, као и добијање што више података.

- Интернет омогућује људима широм света да се брзо повезују и да проналазе велике количине података. Проблеми које при томе треба решавати су проналажење добрих путева којим ће подаци бити усмерени, као и употреба претраживача за брзо проналажење страница на којима се жељена информација налази.

- Електронска трговина омогућује да се роба и услуге размењују електронски. При томе је од суштинског значаја безбедно чување података као што су бројеви кредитних картица, лозинке и бројеви банковних рачуна. Агоритми за шифровање са јавним кључем и дигитални потписи који се за те потребе користе заснивају се на нумеричким алгоритмима и теорији бројева.

## 2.1 Ефикасност алгоритама

Претпоставимо на тренутак да су рачунари бесконачно брзи и да је меморија бесплатна. Да ли би тада било потребе да се проучавају алгоритми? Одговор је - да, у најмању руку зато што треба доказати да се наш метод решавања завршава и да даје коректан одговор.

Ако би рачунари били бесконачно брзи, било који коректан метод за решавање проблема био би прихватљив. И даље би постојао захтев да је реализација алгоритма добра, односно добро документована, али би се обично међу методима бирао онај чија је реализација најједноставнија.

У стварности, рачунари јесу брзи, али не бесконачно брзи. Меморија може бити јефтина, али није бесплатна. Због тога алгоритми морају бити ефикасни у погледу потребног времена и меморијског простора.

Алгоритми намењени решавању истог проблема често се драстично разликују у погледу ефикасности. У наставку ћемо размотрити два алгоритма за сортирање. Трајање извршавања првог од њих — **сортирања уметањем** је $c_1n^2$ ако се сортира низ дужине $n$, где је $c_1$ константа која не зависи од $n$. Другим речима, утрошено време пропорционално је са $n^2$. Трајање извршавања другог алгоритма, **сортирања обједињавањем**, је $c_2n\log n$, где је $\log n = \log_2 n$, а $c_2$ је друга константа, која такође не зависи од $n$. Обично је константа за сортирање уметањем мања од константе за сортирање обједињавањем, односно $c_1 < c_2$. Видећемо да константни фактори много мање утичу на време извршавања него облик зависности од величине улаза $n$. У изразу код сортирања обједињавањем имамо фактор $\log n$, а код сортирања уметањем фактор $n$, који је много већи. Иако је сортирање уметањем обично брже од сортирања обједињавањем за низове мале дужине, кад величина улаза $n$ постане довољно велика, предност сортирања обједињавањем у односу $\log n/n$ довољна је да компензује разлику у константним факторима. Колико год $c_1$ било мање од $c_2$, увек ће постојати гранична вредност $n$, после које ће сортирање обједињавањем бити брже.

Претпоставимо да хоћемо да упоредимо бржи рачунар $A$ на коме се извршава сортирање уметањем и спорији рачунар $B$ на коме се извршава сортирање обједињавањем. Оба рачунара треба да сортирају низ од милион бројева. Нека рачунар $A$ извршава милијарду операција у секунди, а рачунар $B$ само десет милиона операција у секунди, односно да је рачунар $A$ 100 пута бржи од рачунара $B$. Претпоставимо поред тога да је највештији програмер на свету написао програм за сортирање уметањем на машинском језику за рачунар $A$, тако да добијени програм извршава $2n^2$ инструкција при сортирању низа дужине $n$ (односно $c_1 = 2$). С друге стране, програм за сортирање обједињавање за рачунар $B$ написао је осредњи програмер на високом језику, користећи неефикасни преводилац, тако да добијени програм извршава $50n\log n$ инструкција (односно $c_2 = 50$). За сортирање милион бројева рачунару $A$ потребно је

$$\frac{2\cdot(10^6)^2\text{instrukcija}}{10^9\text{instrukcija/sec}} = 2000\text{sec},$$

док је рачунару $B$ потребно

$$\frac{50\cdot 10^6\log 10^6\text{instrukcija}}{10^7\text{instrukcija/sec}} \simeq 100\text{sec}.$$

Користећи алгоритам чије време извршавања спорије расте са $n$, чак и са лошијим преводиоцем, рачунар $B$ је бржи 20 пута од рачунара $A$. Пред-

ност сортирања обједињавањем постаје још значајнија када се сортира десет милиона бројева: тада је за сортирање обједињавањем потребно 2.3 дана, а сортирање обједињавањем завршава се после мање од 20 min. Уопште, са порастом величине низа расте предност сортирања обједињавањем.

# 3 Сортирање уметањем

**Сортирање уметањем** је ефикасан алгоритам за сортирање малог броја елемената. Алгоритам ради на начин на који већина људи ређа у руци карте за играње. Започињемо са празном левом руком и неколико карата на столу, окренутих лицем на доле. Затим узимамо једну по једну карту са стола и умећемо је на њено место у левој руци, као што је то приказано на слици 1. У сваком тренутку су карте у левој руци поређане по величини (сортиране); при томе су то карте које су биле на врху гомиле карата на столу.

Рис. 1: Сортирање карата применом сортирања уметањем

Алгоритам се може описати псеудокодом. Улазни параметри алгоритма су низ $A$ који треба сортирати, и $n$ — дужина низа $A$. Улазни низ се **сортира у месту**: бројеви се премештају у оквиру низа $A$, при чему се највише константан број њих записује ван низа у сваком тренутку. У тренутку завршетка алгоритма низ $A$ садржи сортирани излазни низ.

```
SortiranjeUmetanjem(A, n)
1 for j ← 2 to n do
2    t ← A[j]
3    {уметнути A[j] у сортирани низ A[1..j − 1]}
4    i ← j − 1
5    while i > 0 and A[i] > t do
6       A[i + 1] ← A[i]
7       i ← i − 1
8    A[i + 1] ← t
```

## 3.1 Инваријанта петље и доказ коректности сортирања уметањем

На слици 2. Приказано је како алгоритам ради за низ $A = (5, 2, 4, 6, 1, 3)$. Индекс $j$ је индекс "текуће карте", карте која се тренутно умеће у леву руку. На почетку сваког проласка кроз "спољашњу" **for** петљу, којој одговара индекс $j$, подниз $A[1..j-1]$ одговара картама које се тренутно налазе у левој руци, а елементи $A[j+1..n]$ одговарају картама које се још налазе на столу. Прецизније, елементи $A[1..j-1]$ су уствари елементи који су се на почетку налазили на позицијама од 1 до $j-1$, али су сада поређани по величини. За ту особину подниза $A[1..j-1]$ кажемо да је **инваријанта петље**.

На почетку сваког проласка кроз **for** петљу у линијама $1-8$ подниз $A[1..j-1]$ састоји се од елемената који су на почетку били у поднизу $A[1..j-1]$, али у неопадајућем поретку.

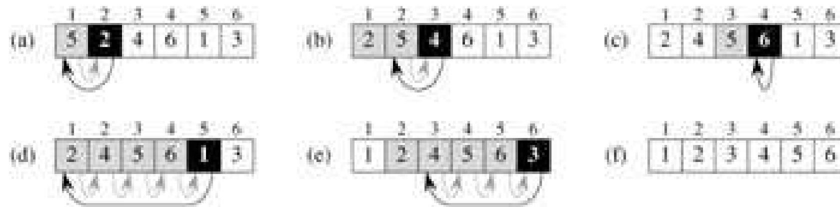


Figure 2: Резултат примене алгоритма СортирањеУметањем на низ $A = (5, 2, 4, 6, 1, 3)$. *(a)-(e)* Итерације **for** петље из линија $1-8$. У свакој итерацији црни квадрат садржи број узет из $A[j]$, који се упоређује са вредностима из сивих квадрата лево од себе у линији 5. Сиве стрелице показују бројеве који су померени за једно место удесно у линији 6, а црна стрелица показује где је број преписан у линији 8. *(f)* Сортирани низ на крају.

Инваријанта петље нам помаже да схватимо зашто је алгоритам коректан. У вези са инваријантом петље треба да докажемо три ствари:

**Иницијализација:** Тврђење је тачно пре првог проласка кроз петљу.

**Одржавање:** Ако је тврђење тачно пре проласка кроз петљу, онда је тачно и пре наредног проласка кроз петљу.

**Завршетак:** По завршетку петље тачност инваријанте петље има за последицу тврђење из кога следи да је алгоритам коректан.

Кад су прва два тврђења тачна онда је инваријанта петље тачна пре сваког проласка кроз петљу, на основу принципа математичке индукције. Треће тврђење је коначан доказ коректности алгоритма. Погледајмо ове три особине у случају сортирања уметањем.

**Иницијализација:** На почетку треба показати да је тврђење тачно пре првог проласка кроз петљу, кад је $j = 2$. Тада се подниз $A[1..j-1]$ састоји само од елемента $A[1]$, који је уствари првобитни елеменат $A[1]$. Поред тога, подниз је (тривијално) уређен неопадајуће. Према томе, инваријанта петље је тачна пре првог проласка кроз петљу.

**Одржавање:** Сада треба доказати да сваки наредни пролазак кроз петљу задржава тачност инваријанте петље. Заиста, у телу спољашње **for** петље елементи $A[j-1]$, $A[j-2]$, $A[j-3]$, $\ldots$ премештају се по једно место удесно, све док се не нађе право место за $A[j]$ (линије $4-7$). Тада се вредност $A[j]$ копира на то место (линија 8).

**Завршетак:** Испитајмо шта се дешава по завршетку петље. Спољашња петља се завршава кад $j$ постане веће од $n$, тј. кад је $j = n+1$. Замењујући $j$ са $n+1$ у инваријанти петље, добијамо да се подниз $A[1..n]$ састоји од елемената који су у почетку били у $A[1..n]$, али у неопадајућем поретку. Међутим, подниз $A[1..n]$ је уствари цео низ! Према томе, цео низ је сортиран, што значи да је алгоритам коректан.

## 3.2 Псеудокод

За специфицирање алгоритма искористили смо специјални неформални језик, **псеудокод**. Наводимо нека од правила овог језика.

1. Увлачење редова одговара блоковској структури. На пример, **for** петља која почиње од линије 1, састоји се од линија $2-8$; тело **while** петље која почиње у линији 5 садржи линије $6-7$, али не и линију 8. Увљачење линија примењује се на уобичајени начин и на **if** $-$ **then** $-$ **else** наредбе. Сврха увлачења редова је повећање јасноће кода.

2. Петље **while** и **for** , као и условне наредбе **if** , **then** , **else** имају значење слично као у Паскалу. Изузетак је да у **for** петљи бројачка променљива после проласка кроз петљу задржава своју вредност. Тако, кад се **for** петља из линије 1 (**for** $j \leftarrow 2$ **to** $n$ **do** ) заврши, биће $j = n+1$.

3. Текст у витичастим заградама је коментар.

4. Члан низа означава се додавањем индекса у угластим заградама. На пример, $A[i]$ је $i$-ти члан низа $A$. Ознака .. служи за издвајање подниза. Тако $A[1..j]$ означава подниз низа $A$ који се састоји од $j$ елемената $A[1]$, $A[2]$,$\ldots$,$A[j]$.

# 4 Анализа алгоритама

**Анализа** алгоритма подразумева предвиђање ресурса потребних за његово извршавање, и то најчешће времена, односно меморијског простора. Уопште, анализирајући неколико алгоритама — кандидата за решавање датог проблема, обично се лако издваја најефикаснији. Резултат анализе може бити да постоји више од једног прихватљивог кандидата, док се више лошијих алгоритама одбацују.

Претпоставља се да се алгоритам извршава као рачунарски програм, инструкција по инструкција, без истовремених операција.

Прецизнија анализа захтевала би дефинисање скупа инструкција на рачунару и њихова трајања. То би било превише напорно и не би дало много информација потребних при анализи и конструкцији алгоритама. С друге стране, не сме се злупотребити модел рачунара — шта ако бисмо претпоставили да рачунар има инструкцију за сортирање? Таква претпоставка је, наравно, нереална. Дакле, претпостављаћемо да рачунар може да извршава инструкције које постоје на обичним рачунарима: аритметичке (сабирање, одузимање, множење, дељење), копирање података, инструкције за контролу тока (условни и безусловни скок, позив потпрограма и повратак из њега).

Анализа чак и једноставих алгоритама у оваквом моделу може да представља изазов. Потребни математички алати су комбинаторика, теорија вероватноће, манипулација са алгебарским изразима и способност препознавања најзначајнијих чланова у изразу. Због тога што се алгоритам може понашати другачије за сваки могући улаз, потребно је имати на располагању средства за представљање њиховог понашања једноставним, лако разумљивим изразима.

## 4.1 Анализа сортирања уметањем

Време потребно за СортирањеУметањем зависи од улаза: сортирање хиљаду бројева траје дуже од сортирања три броја. Поред тога, СортирањеУметањем може да траје различито ако се сортирају два низа исте дужине, зависно од степена колико су они већ сортирани. У општем случају, време које троши алгоритам расте са величином улаза, па је уобичајено да се време извршавања програма изражава у зависности од величине улаза. При томе се "време извршавања" и "величина улаза" морају пажљивије дефинисати.

Појам **величине улаза** зависи од проблема који се решава. За многе проблеме, као што је сортирање, најприроднија мера је *број елемената у улазу* — на пример дужина $n$ низа који се сортира. За многе друге проблеме, као што је на пример множење два броја, то је *укупан број бита* потребних да се улаз представи у бинарном облику.

**Време извршавања** алгоритма за конкретан улаз је број примитивних операција или "корака" који се извршавају. Уобичајено је да се појам корака дефинише тако да што мање зависи од конкретног рачунара. У нашим примерима применићемо следећи приступ. За извршавање сваке линије псеудокода потребно је константно време. Извршавање једне линије захтева другачије време од извршавања друге линије, али ћемо претпоставити да извршавање $i$-те линије траје $c_i$, где је $c_i$ константа.

У наставку ће израз за време извршавања алгоритма СортирањеУметањем еволуирати из компликоване формуле која користи све константне цене $c_i$ у много једноставнији израз коришћењем једноставније нотације. Та једноставнија нотација ће такође омогућити утврђивање да ли је један алгоритам ефикаснији од другог.

Најпре процедуру СортирањеУметањем допуњујемо оценама трајања инструкција (линија). За свако $j = 2, 3, \dots n$ нека $t_j$ означава број колико пута је извршен тест у **while** петљи у линији 5 за ту вредност $j$. Коментари се не извршавају, па не троше време.

| $SortiranjeUmetanjem(A, n)$ | цена | број понављања |
|---|---|---|
| 1 **for** $j \leftarrow 2$ **to** $n$ **do** | $c_1$ | $n$ |
| 2 $\quad t \leftarrow A[j]$ | $c_2$ | $n-1$ |
| 3 $\quad$ {уметнути $A[j]$ у сортирани низ $A[1..j-1]$} | 0 | $n-1$ |
| 4 $\quad i \leftarrow j-1$ | $c_4$ | $n-1$ |
| 5 $\quad$ **while** $i > 0$ **and** $A[i] > t$ **do** | $c_5$ | $\sum_{j=2}^{n} t_j$ |
| 6 $\quad\quad A[i+1] \leftarrow A[i]$ | $c_6$ | $\sum_{j=2}^{n}(t_j-1)$ |
| 7 $\quad\quad i \leftarrow i-1$ | $c_7$ | $\sum_{j=2}^{n}(t_j-1)$ |
| 8 $\quad A[i+1] \leftarrow t$ | $c_8$ | $n-1$ |

Време извршавања алгоритма је збир времена извршавања свих инструкција. Ако се нека инструција састоји од $c_i$ корака, а извршава се $n$ пута, онда је њен допринос укупном времену $c_i n$. Да бисмо израчунали

$T(n)$, време извршавања алгоритам СортирањеУметањем, треба да саберемо производе у колонама *цена* и *број понављања*, па је

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + \\ &+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1). \end{aligned}$$

Чак и за улазе задате величине, време извршавања алгоритма може да зависи од тога *који* улаз те величине је задат. На пример, за алгоритам СортирањеУметањем најбољи случај је кад је низ већ сортиран. За свако $j = 2, 3, \ldots, n$ испоставља се да је $A[i] \le t$ у линији 5 тачно кад $i$ има своју почетну вредност $j-1$. Према томе, $t_j = 1$ за $j = 2, 3, \ldots, n$, па је време извршавања у најбољем случају

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Ово време извршавања може се изразити у облику $an + b$ за неке константе $a$ и $b$, које зависе од цена наредби $c_j$; време извршавања је дакле **линеарна функција** од $n$.

Ако је низ на почетку опадајући, добија се најгори случај. Елеменат $A[j]$ мора се упоредити са сваким елементом подниза $A[1..j-1]$, па је $t_j = j$ за $j = 2, 3, \ldots, n$. Пошто је $\sum_{j=2}^{n} j = n(n+1)/2 - 1$ и $\sum_{j=2}^{n} (j-1) = n(n-1)/2$, добијамо да је у најгорем случају време извршавања алгоритма СортирањеУметањем једнако

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left( \frac{n(n+1)}{2} - 1 \right) \\ &+ c_6 \left( \frac{n(n-1)}{2} \right) + c_7 \left( \frac{n(n-1)}{2} \right) + c_8(n-1) \\ &= \left( \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left( c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\ &\quad -(c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Ово време извршавања може се изразити у облику $an^2 + bn + c$ за неке константе $a$, $b$ и $c$ које зависе од цена наредби $c_j$; време извршавања је дакле **квадратна функција** од $n$.

## 4.2 Анализа најгорег и просечног случаја

У спроведеној анализи сортирања уметањем разматрали смо и најбољи случај, кад је низ већ сортиран, и најгори случај, кад је низ сортиран обр-

нутим редоследом, опадајуће. Приликом анализе алгоритама обично се разматра само **време извршавања у најгорем случају**. Постоји неколико аргумената за овакав приступ.

- Време извршавања алгоритма у најгорем случају је горња граница за време извршавања алгоритма за било који улаз исте величине. Знајући то, имамо гаранцију да извршавање алгоритма никада неће трајати више од процењеног времена.

- За неке алгоритме се најгори случај дешава прилично често. На пример, приликом тражења конкретног податка у бази података, најгори случај је кад тог податка нема у бази. У неким применама претраживања база података тражење одсутног податка може да буде чест случај.

- "Просечан случај" је често исто толико лош колико и најгори случај. Претпоставимо да на случајан начин изаберемо $n$ бројева и применимо на њих сортирање уметањем. Колико треба времена да се одреди где је у поднизу $A[1..j-1]$ место за уметање елемента $A[j]$? У просеку је пола елемената $A[1..j-1]$ мање од $A[j]$, а пола је веће од њега. У просеку треба проверити пола чланова низа $A[1..j-1]$, па је $t_j = j/2$. Ако ову анализу просечног случаја спроведемо до краја, добијамо да је просечно време извршавања такође квадратна функија од $n$, као и у најгорем случају.

## 4.3 Брзина раста

Приликом анализе сортирања уметањем користили смо нека упрошћавања засnoвана на уопштавању. Најпре, занемаривали смо стварне цене појединих инструкција, коришћењем константи $c_i$ за представљање њихових цена. Затим смо запазили да нам те константе пружају више детаља него што нам је потребно: време извршавања је $an^2 + bn + c$ за неке константе $a$, $b$ и $c$ које зависе од цена инструкција $c_i$. Према томе, ми смо игнорисали не само стварне цене инструкција, него и апстрактне цене $c_i$.

Сада ћемо учинити следеће упрошћење засновано на уопштавању. Ради се о томе да нас занима само **брзини раста** односно **поредак раста** времена извршавања. Због тога ми разматрамо само водећи члан (нпр. $an^2$), јер чланови нижег реда постају занемарљиви за велике вредности $n$. Према томе, ми кажемо да, на пример, сортирање уметањем има време извршавања $\Theta(n^2)$ у најгорем случају. Ову ознаку користићемо неформално, иако је лако дефинисати је и прецизно. Обично сматрамо један алгоритам ефикаснијим од другог ако његово време извршавања има мању брзину раста. Збок константних фактора и занемарених чланова нижег реда, ова процена може бити погрешна за мале $n$. Међутим, за довољно велике улазе се на пример $\Theta(n)$ алгоритам извршава брже од $\Theta(n^2)$ алгоритма.

# 5 Конструкција алгоритама применом разлагања

Постоји више техника за конструкцију алгоритама. Код сортирања уметањем коришћен је **инкрементални** приступ: полазећи од сортираног подниза $A[1..j-1]$, ми умећемо наредни елеменат $A[j]$ на одговарајуће место, и тако долазимо до сортираног подниза $A[1..j]$.

Сада ћемо размотрити други приступ конструкцији алгоритама, заснован на разлагању улазних података (енглески divide-and-conquer). Користећи овај приступ, долази се до алгоритма за сортирање чије је време извршавања много мање него за сортирање уметањем. Алгоритми засновани на овом приступу имају предност да се њихово време извршавања лако одређује у веома општем случају.

Многи корисни алгоритми имају **рекурзивну** структуру: да би решили дати проблем, они позивају себе да би обрадили блиско повезане потпроблеме. Такви алгоритми често користе приступ заснован на **разлагању**: они разбијају проблем на неколико сличних, али мањих потпроблема, решавају их рекурзивно, а онда комбинују ова решења да би формирали решење полазног проблема.

Приступ заснован на разлагању укључује три корака на сваком нивоу рекурзије:

**Разлагање** проблема на неколико мањих потпроблема.

Рекурзивно **решавање потпроблема**. У случају кад је величина потпроблема довољно мала, потпроблем се решава директно, тј. излази се из рекурзије.

**Комбиновање** решења потпроблема да би се добило решење полазног проблема.

Алгоритам **сортирање обједињавањем** директно реализује ову технику. Интуитивно, он се извршава на следећи начин.

**Разлагање:** Низ дужине $n$ који треба сортирати дели се на два подниза дужине по $n/2$.

**Решавање потпроблема:** Два подниза се рекурзивно сортирају применом истог алгоритма.

**Комбиновање:** Два добијена сортирана подниза се обједињавају, и тако се добија сортирани полазни низ.

Из рекурзије се излази кад поднизови које треба сортирати имају дужину 1: такви низови су већ сортирани.

Кључна операција у овом алгоритму је обједињавање два сортирана низа у кораку "комбиновање". Да бисмо извршили обједињавање, користимо помоћну процедуру $OBJEDINJAVANJE(A, p, q, r)$, где је $A$ низ, а $p$, $q$ и $r$ су

индекси чланова низа $A$, такви да је $p \leq q < r$. Процедура претпоставља да су поднизови $A[p..q]$ и $A[q+1..r]$ већ сортирани. Процедура их **обједињује** у један сортирани подниз, који замењује текући подниз $A[p..r]$.

Процедура ОБЈЕДИЊАВАЊЕ извршава се за време $\Theta(n)$, где је $n = r - p + 1$ број елемената који се обједињавају, и ради на следећи начин. Враћајући се на мотив са картама, претпоставимо да су на столу две гомиле карата лицем окренутим навише. Обе гомиле су сортиране, тако да је најмања карта на врху. Наш циљ је да објединимо две гомиле у једну сортирану излазну гомилу, у којој ће карте бити окренуте лицем наниже. Основни корак је избор мање од две карте на врху гомила, скидање те карте са гомиле (чиме се открива наредна карта на истој гомили) и њено пребацивање лицем наниже на излазну гомилу. Основни корак се понавља све док се не испразни једна од улазних гомила; тада се друга улазна гомила пребацује лицем наниже на излазну гомилу. За извршавање сваког оваквог корака потребно је константно време, јер се упоређују само две карте на врху гомила. Пошто је број основних корака који се извршавају највише $n$, време извршавања обједињавања је $\Theta(n)$.

Наредни псеудокод реализује горњу идеју, уз допунски трик који омогућује да се избегне потреба да се у сваком основном кораку проверава да ли је једна од гомила испражњена. Идеја је да се на дно обе гомиле карата стави **граничник**, карта која садржи специјалну вредност, коју користимо да бисмо упростили код. Овде као вредност граничника користимо $\infty$, тако да кад је та карта на врху гомиле, она никад не може бити мања од друге карте, сем ако се и на врху друге гомиле налази та вредност. Али када се то деси, све остале карте сем граничника су већ пребачене на излазну гомилу. Пошто ми унапред знамо да на излазну гомилу треба пребацити тачно $r - p + 1$ карата, заустављамо се у тренутку кад извршимо толико основних корака.

```
OBJEDINJAVANJE(A, p, q, r)
1   n₁ ← q − p + 1
2   n₂ ← r − q
3   креирати низове L[1..n₁ + 1] и D[1..n₂ + 1]
4   for i ← 1 to n₁ do
5      L[i] ← A[p + i − 1]
6   for j ← 1 to n₂ do
7      D[j] ← A[q + j]
8   L[n₁ + 1] ← ∞
9   D[n₂ + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r do
13     if L[i] ≤ D[j] then
14        A[k] ← L[i]
15        i ← i + 1
16     else A[k] ← D[j]
17        j ← j + 1
```

Процедура ОБЈЕДИЊАВАЊЕ ради на следећи начин. У линији 1 израчунава се дужина $n_1$ подниза $A[p..q]$, а у линији 2 дужина $n_2$ подниза $A[q+1..r]$. Затим се креирају низови $L$ и $D$ ("леви" и "десни"), дужина $n_1+1$, односно $n_2+1$, у линији 3. Петља **for** у линијама $4-5$ копира подниз $A[p..q]$ у $L[1..n_1]$, а петља **for** у линијама $6-7$ копира подниз $A[q+1..r]$ у $D[1..n_2]$. Линије $8-9$ уписују граничнике у низове $L$ и $D$. Линије $10-17$, као што је приказано на сликама 3. и 4. извршавају $r-p+1$ основних корака, одржавајући следећу инваријанту петље:

На почетку сваког проласка кроз **for** петљу у линијама $12-17$ подниз $A[p..k-1$ садржи $k-p$ најмањих елемената низова $L[1..n_1+1]$ и $D[1..n_2+1]$, и то у сортираном поретку. Поред тога, $L[i]$ и $D[j]$ су најмањи елементи у одговарајућим низовима међу елементима који још нису копирани назад у $A$.

Показаћемо сада да је инваријанта петље тачна пре првог проласка кроз **for** петљу у линијама $12-17$, да инваријанта остаје тачна после сваког проласка кроз петљу, и да је последица инваријанте тврђење из кога следи коректност алгоритма кад се петља заврши.

**Иницијализација:** Пре првог проласка кроз петљу је $k=p$, па је подниз $A[p..k-1$ празан. Овај празан подниз садржи $k-p=0$ најмањих елемената из $L$ и $D$, па пошто је $i=j=1$, $L[i]$ и $D[j]$ су најмањи елементи у својим низовима који нису били копирани назад у $A$. Тврђење је тачно пре првог проласка кроз петљу.

**Одржавање:** Да бисмо се уверили да сваки пролазак кроз петљу одржава инваријанту петље, претпоставимо најпре да је $L[i] \leq D[j]$. Тада је $L[i]$ најмањи елеменат међу онима који још нису копирани назад у

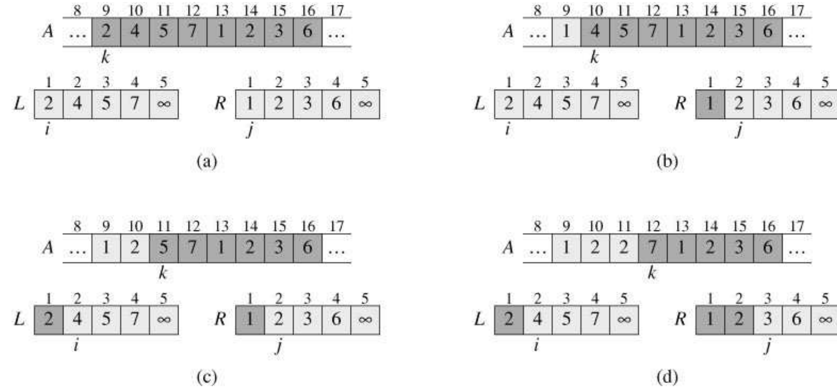


Figure 3: Извршавање линија $10-17$ приликом позива $OBJEDINJAVANJE(A, 9, 12, 16)$ када подниз $A[9..16]$ садржи низ $(2, 4, 5, 7, 1, 2, 33, 6)$. После копирања и уметања граничника садржај низа $L$ је $(2, 4, 5, 7, \infty)$, а садржај низа $D$ је $(1, 2, 3, 6, \infty)$. Светло обојени елементи низа $A$ садрже своје коначне вредности које треба копирати назад у $A$. Посматране заједно, светло обојене позиције у сваком тренутку садрже вредности које су на почетку биле у $A[9..16]$, заједно са два граничника. Затамњене позиције у $A$ садрже вредности које ће бити преписане новим садржајем, а затамњене позиције у $L$ и $D$ садрже вредности које су већ биле прекопиране назад у $A$. $(a)-(h)$ Низови $A$, $L$ и $D$ и одговарајући индекси $k$, $i$ и $j$ пре сваког проласка кроз петљу у линијама $12-17$. $(j)$ Низови и индекси приликом завршетка. У том тренутку је подниз $A[9..16]$ сортиран, а два граничника у $L$ и $D$ у једини елементи у овим низовима који још нису копирани назад у $A$.

$A$. Пошто $A[p..k-1]$ садржи $k-p$ најмањих елемената, када се у линији 14 $L[i]$ прекопира у $A[k]$, подниз $A[p..k]$ садржаће $k-p+1$ најмањих елемената. Повећавање за 1 вредности $k$ (приликом повратка у **for** петљи) и $j$ (у линији 15) обнавља тачност инваријанте петље за следећи пролазак. Ако је пак $L[i] > D[j]$, онда линије $16-17$ извршавају оно што је неопходно да се одржи инваријанта петље.

**Завршетак:** На крају је $k = r+1$. На основу инваријанте петље подниз $A[p..k-1]$, што је уствари $A[p..r]$, садржи $k-p = r-p+1$ најмањих елемената из $L[1..n_1+1]$ и $D[1..n_2+1]$, и то у сортираном поретку. Низови $L$ и $D$ садрже заједно $n_1+n_2 = r-p+3$ елемената. Сви сем два од њих копирају се назад у $A$, а та два највећа елемента су граничници.

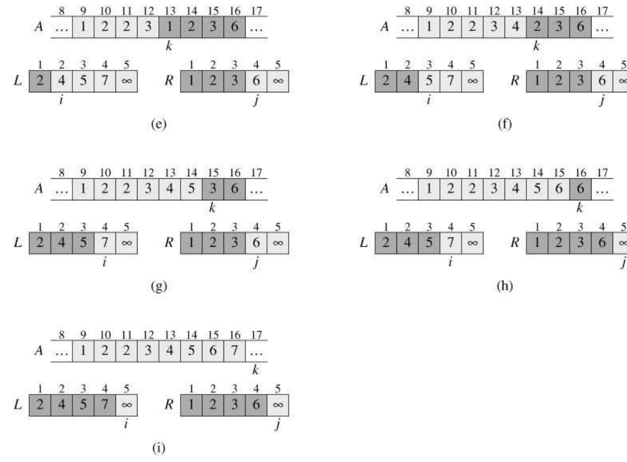


Figure 4: Извршавање линија $10-17$ приликом позива $OBJEDINJAVANJE(A, 9, 12, 16)$, други део.

Да бисмо се уверили да је трајање процедуре ОБЈЕДИЊАВАЊЕ $\Theta(n)$, где је $n = r - p + 1$, приметимо да се линије $1-3$ и $8-11$ извршавају за константно време, да извршавање **for** петљи у линијама $4-7$ траје $\Theta(n_1 + n_2) = \Theta(n)$, као и да се кроз **for** петљу у линијама $12-17$ пролази $n$ пута, при чему сваки пролазак троши константно време.

Сада можемо да искористимо процедуру ОБЈЕДИЊАВАЊЕ као потпрограм у алгоритму за сортирање обједињавањем. Процедура

$$SortiranjeObjedinjavanjem(A, p, r)$$

сортира елементе у поднизу $A[p..r]$. Ако је $p \geq r$ онда подниз има највише један елемент, па је према томе већ сортиран. У противном, корак разлагања једноставно израчунава индекс $q$, који раздваја $A[p..r]$ у два подниза: $A[p..q]$ са $\lceil n/2 \rceil$ елемената, и $A[q+1..r]$ са $\lfloor n/2 \rfloor$ елемената. Овде $\lceil x \rceil$, односно $\lfloor x \rfloor$ означавају редом најмањи цели број већи или једнак од $x$, односно највећи цели број мањи или једнак од $x$.

$SortiranjeObjedinjavanjem(A, p, r)$
1  **if** $p < r$ **then**
2   $q \leftarrow \lfloor (p+r)/2 \rfloor$
3   $SortiranjeObjedinjavanjem(A, p, q)$
4   $SortiranjeObjedinjavanjem(A, q+1, r)$
5   $Objedinjavanje(A, p, q, r)$

Да бисмо извршили сортирање целог низа $A = (A[1], A[2], \ldots, A[n])$, позивамо $SortiranjeObjedinjavanjem(A, 1, n)$. Слика 5. илуструје извршавање процедуре одоздо навише када је $n$ степен двојке. У алгоритму се обједињавају парови 1-чланих поднизова у двочлане поднизове, затим се

обједињавају парови поднизова дужине 2 у поднизове дужине 4, и тако даље, све док се на крају два подниза дужине $n/2$ не обједине у низ дужине $n$.

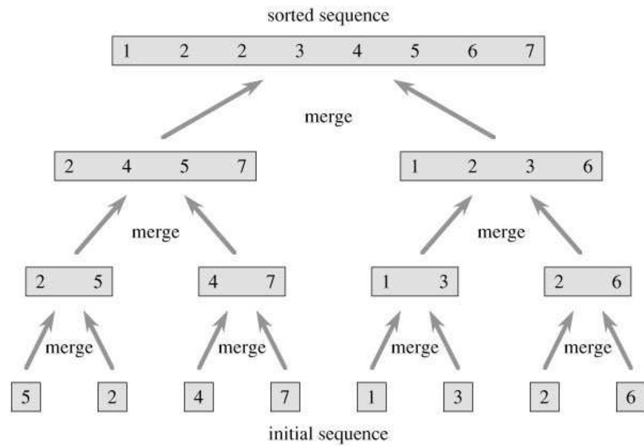


Figure 5: Резултат примене сортирања обједињавањем на низ $A = (5, 2, 4, 7, 1, 3, 2, 6)$. Дужине сортираних поднизова који се обједињавају расту са напредовањем алгоритма одоздо навише.

# 6 Анализа алгоритама заснованих на разлагању

Када алгоритам садржи рекурзивни позив самог себе, његово време извршавања се често може описати **рекурентном релацијом**, која изражава укупно време извршавања на проблему са улазом величине $n$ преко времена извршавања на мањим улазима. Тада се овакве рекурентне релације могу решити да би се добила граница за ефикасност алгоритма.

Рекурентна релација за време извршавања алгоритма заснованог на разлагању одређена је са три основна корака оваквог лагоритма. Нека, као и раније, $T(n)$ означава време извршавања на улазу величине $n$. Ако је величина улаза довољно мала, нпр. $n \leq c$ за неку константу $c$, директно решавање траје константно време, што се може написати као $\Theta(1)$. Претпоставимо да се разлагањем улаза добија $a$ потпроблема, од којих је сваки $b$ пута мањи од полазног. (У алгоритму сортирања обједињавањем је $a = b = 2$, али постоје примери оваквих алгоритама у којима је $a \neq b$). Ако са $D(n)$ означимо време потребно да се улаз разложи, а са $C(n)$ да се комбинују решења потпроблема у решење полазног проблема, добијамо рекурентну релацију

$$T(n) = \begin{cases} \Theta(1), & n \leq c, \\ aT(n/b) + D(n) + C(n), & n > c. \end{cases}$$

## 6.1 Анализа сортирања обједињавањем

Иако псеудокод за сортирање обједињавање исправно ради и ако дужина низа није парна, анализа решавањем рекурентне релације се поједностављује ако се претпостави да је почетна дужина низа степен двојке.

Сада ћемо извести рекурентну релацију за $T(n)$, оцену времена извршавања сортирања обједињавањем у најгорем случају. Сортирање једночланог низа траје константно време. Ако треба сортирати $n > 1$ елемената, време извршавања разлажемо на следећи начин.

**Разлагање:** Овај корак израчунава средину низа, па се извршава за константно време. Дакле, $D(n) = \Theta(1)$.

**Решавање потпроблема:** Два подниза дужине $n/2$ се рекурзивно сортирају применом истог алгоритма, што траје $2T(n/2)$.

**Комбиновање:** Видели смо да извршавање процедуре ОБЈЕДИЊАВАЊЕ на подниз дужине $n$ траје $\Theta(n)$, па је $C(n) = \Theta(n)$.

Сабирајући функције $D(n)$ и $C(n)$ приликом анализе сортирања обједињавања, ми сабирамо једну функцију која је $\Theta(n)$ са једном функцијом која је $\Theta(1)$. Збир је линеарна функција од $n$, односно $\Theta(n)$. Према томе, тражена рекурентна релација је

$$T(n) = \begin{cases} \Theta(1), & n = 1, \\ 2T(n/2) + cn, & n > 1. \end{cases}$$

где је $c$ утрошено време по елементу низа у корацима разлагања и комбиновања.

Слика 6. показује како се може решити ова рекурентна релација. Због једноставности, претпоставимо да је $n$ степен двојке. Део $(a)$ слике приказује $T(n)$, израз који је у делу $(b)$ развијен у еквивалентно стабло које представља рекурентну релацију. Члан $cn$ је у корену (трајање највишег нивоа рекурзије), а два подстабла су трајања два мања рекурентна позива $T(n/2)$. Део $(c)$ показује резултат примене истог поступка после развоја $T(n/2)$. Трајање сваког чвора на другом нивоу рекурзије је $cn/2$ Настављамо са развијањем сваког чвора стабла његовим разбијањем на саставне делове одређене рекурентном релацијом, све док величине поднизова не дођу до 1. Део $(d)$ показује коначно стабло.

После тога ми сабирамо цене на сваком нивоу стабла. Највиши ниво има цену $cn$; следећи ниво има цену $c(n/2) + c(n/2) = cn$ следећи ниво има цену $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$, итд. Уопште, ниво $i$ испод корена има $2^i$ чворова са ценом $c(n/2^i)$, па је укупна цена свих чворова на $i$-том нивоу испод врха једнака $2^i c(n/2^i) = cn$. На најнижем нивоу има $n$ чворова са ценом $c$, па је њихова укупна цена $cn$.

Укупан број нивоа у "стаблу рекурзије" на слици 5. је $\log n + 1$, што се лако показује индукцијом. Да бисмо добили решење рекурентне релације, треба да саберемо цене на свим нивоима овог стабла. Укупно има $\log n +$

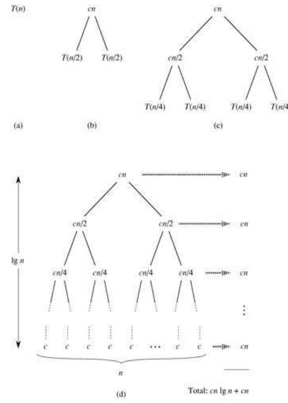


Figure 6: Конструкција стабла рекурзије за рекурентну релацију $T(n) = 2T(n/2) + cn$. Део $(a)$ приказује $T(n)$ који се прогресивно разлаже у деловима $(b) - (d)$ да би се добило стабло рекурзије. Потпуно развијено стабло рекурзије у делу $(d)$ има $\log n + 1$ нивоа (односно има висину $\log n$, као што је назначено), а збир времена у свим чворовима на истом нивоу је $cn$. Укупна сума је, према томе, $cn \log n + cn$, што је $\Theta(n \log n)$.

1 нивоа са ценом по $cn$, па је укупна цена $cn(\log n + 1) = cn \log n + cn$. Занемарујући члан нижег реда и константу $c$, добијамо резултат $T(n) = \Theta(n \log n)$.

# 7 Литература

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, Second Edition, MIT Press, 2002.