Pregled i usporedba algoritama za pretraživanje

Kanceljak, Marko

VELEUČILIŠTE
s pravom javnosti
BALTAZAR ZAPREŠIĆ
Zaprešić

Prijediplomski stručni studij

Informacijske tehnologije

MARKO KANCELJAK

PREGLED I USPOREDBA ALGORITAMA ZA PRETRAŽIVANJE

PRIJEDIPLOMSKI ZAVRŠNI RAD

Zaprešić, 2023. godine

VELEUČILIŠTE
s pravom javnosti
BALTAZAR ZAPREŠIĆ
Zaprešić

Prijedipomski stručni studij
Informacijske tehnologije

PRIJEDIPLOMSKI ZAVRŠNI RAD

PREGLED I USPOREDBA ALGORITAMA ZA PRETRAŽIVANJE

Mentor:
dr. sc. Alisa Bilal Zorić

Student:
Marko Kanceljak

Naziv kolegija:
Strukture podataka i algoritmi

JMBAG studenta:
0016133603

Sadržaj

SAŽETAK 1
Abstract 2
1. UVOD 3
2. OSNOVNI POJMOVI 4
2.1 Algoritmi i strukture podataka 4
2.2 Lista 4
3. ALGORITMI SORTIRANJA 6
3.1 Algoritmi pretraživanja kroz povijest 7
3.2 VREMENSKO TRAJANJE ALGORITMA 9
4. ALGORITMI PRETRAŽIVANJA 10
4.1 Algoritmi sekvencijalnog pretraživanja 10
4.2 Linearno pretraživanje 10
4.2.1 Vremenska kompleksnost linearnog pretraživanja 12
4.3 Algoritmi intervalnog pretraživanja 13
4.3.1 Primjer binarnog pretraživanja 13
4.3.2 Jump search algoritam 15
4.3.3 Interpolation search algoritam 18
4.3.4 Eksponencijalno pretraživanje 19
4.3.5 Fibonacci search algoritam 22
4.3.6 Hash Table Search Algorithm 25
5. Usporedba algoritama pretraživanja 27
5.1 Linearno pretraživanje – usporedba 27
5.2 Binarno pretraživanje – usporedba 27
5.3 Jump search – usporedba 28
5.4 Interpolation search - usporedba 28
5.5 Eksponential search – usporedba 28
5.6 Fibonacci search – usporedba 29
6. Algoritmi pretraživanja u praksi 29
6.1 Problem trgovačkog putnika (eng. Traveling salesman problem) 30
6.2 Google algoritam pretraživanja 30
7. Pregled navedenih algoritama 31

7.1 Linearno pretraživanje – primjer 31
7.2 Binarno pretraživanje - primjer 31
7.3 Jump search – primjer 32
7.4 Interpolation search algoritam 33
7.5 Eksponencijalno pretraživanje 34
7.6 Fibonacci search algoritam 35
8. Zaključak 36
9. Izjava 37
10. Literatura 38
11. Popis slika 39
12. Popis tablica 39

SAŽETAK

Kroz rad će biti opisani algoritmi pretraživanja, čemu služe, kako i zašto su nastali, njihov razvoj kroz povijest, te će se opisati neki primjeri iz prakse. Nakon toga će biti prikazana implementacija algoritama pretraživanja u programskom jeziku Phyton, odnosno analizirat će se i usporediti više vrsta i načina pretraživanja (neki od algoritama pretraživanja koje će se analizirati u radu su: Linearno pretraživanje, Binarno pretraživanje, Interpolacijsko pretraživanje, Fibonaccijevo pretraživanje, te će se pobliže vizualno objasniti pojedini koraci u algoritmima.

Rad će se posebno fokusirati na efikasnost pojedinih algoritama, odnosno vrijeme izvođenja, kompleksnost izvođenja, idealan slučaj i najgori slučaj, te prednosti i nedostatke istih. Svaki algoritam će imati identične ulazne parametre (listu nazvanu *li* koja se sastoji od niza prirodnih brojeva. Odnosno: *li*[1, 2, 3, 4, ... ,10^n] gdje je n prirodan broj. Iz zadane liste računalno nasumično izabire jedan broj uz pomoć *rand()* funkcije.

Svaki od algoritama traži zadani broj na svoj specifičan način pri čemu se mjeri vrijeme izvođenja u sekundama i kompleksnost trenutnog algoritma, te se nadalje detaljno proučava.

Ključne riječi: algoritam, pretraživanje, lista

Abstract

The paper will describe search algorithms, what they are for, how and why they were created, their development through history, and we will give some examples in practice. After that, the implementation of search algorithms in the Python programming language will be presented, that is, we will analyze and compare several types and methods of search (some of the search algorithms that we will analyze in the paper are: Linear search, Binary search, Interpolation search, Fibonacci search, and we will visually explain individual steps in algorithms.

The work will focus especially on the efficiency of individual algorithms, i.e. execution time, execution complexity, ideal case and worst case, as well as their advantages and disadvantages. Each algorithm will have identical input parameters (a list named or consisting of a series of natural numbers. That is: li[1, 2, 3, 4, ... ,10n] where n is a natural number. From the given list, you randomly choose one number using the rand() function. Each of the algorithms searches for a given number in its own specific way, measuring the execution time in seconds and the complexity of the current algorithm, and is further studied in detail.

Key words: algorithm, search, list

1. UVOD

U današnje vrijeme i uz današnju količinu podataka bilo bi nemoguće snaći se bez pravilno sortiranih podataka. Tek nakon što su podatci pravilno sortirani možemo raditi s njima odgovarajuće zadatke. Kako bi ekstrahirali korisne informacije od dobivene količine podataka koristimo se algoritmima za pretraživanje. Cilj algoritma pretraživanja je dostaviti tražene podatke krajnjem korisniku u što kraćem vremenu. Naravno, različiti algoritmi drugačijih kompleksnosti imaju različita vremena izvođenja, odnosno neki su kompliciraniji, neki su jednostavniji, neki su brži neki su sporiji.

Kako navodi Jenifer Mosher u svojem radu tražiti znači “pogledati ili ispitati pažljivo kako bi pronašli nešto skriveno.” (J. Mosher, 2010.) Algoritam je "skup pravila za rješavanje problema u konačnom broju koraka." Dakle, algoritam pretraživanja je promatranje ili ispitivanje problema, sa skupom pravila za rješavanje navedenog problema, u konačnom broju koraka.

U ovom radu će algoritmi pretraživanja biti prikazani u programskom jeziku Phyton. Tema ovog rada je pretraživanje podataka, samim time ćemo se više posvetiti pretraživanju. Važno je naglasiti da se ne može pretraživati bilo kakav skup podataka nego samo sortirani, bilo to uzlazno, od najmanjeg prema najvećem ili silazni, od najvećeg prema najmanjem. Mi ćemo u ovom radu raditi s već sortiranom listom brojeva. Ulazni parametar svakog algoritma pretraživanja biti će sortirana lista brojeva. Lista nazvana *li* [0, 1, 2, 3, ..., 10^n] gdje je n prirodan broj. Iz zadane liste *li* ćemo nasumično izabrati jedan broj te uz različite algoritme pretraživanja „tražiti“ zadani broj uz više algoritama pretraživanja, te ćemo proučavati vrijeme izvođenja, kompleksnost, idealan i najgori slučaj.

2. OSNOVNI POJMOVI

Na početku rada ćemo definirati osnovne pojmove vezane za programiranje kao što su: instrukcija, algoritam, strukture podataka, i slično. U Hrvatskoj enciklopediji nalazimo definiciju koja vrlo dobro objašnjava pojmove koje nalazimo u radu: „Instrukcija (u programskom kontekstu) je dio računalnoga programa kojim se propisuje sljedeća operacija.“ Jednostavnije rečeno instrukcijom govorimo računalu što treba napraviti. Računalo ne zna ništa napraviti samostalno te sve shvaća kroz niz instrukcija koje su unaprijed propisane. Instrukcije se izvršavaju redoslijedom kojim su napisane, od prve prema zadnjoj, liniju po liniju.

2.1 Algoritmi i strukture podataka

Algoritam je skup simbola i općeniti postupak za sustavno rješavanje pojedinačnih zadataka iz neke određene klase matematičkih problema. (Hrvatska enciklopedija, 2022). npr. Euklidov algoritam za pronalaženje najveće zajedničke mjere dvaju prirodnih brojeva, Eratostenovo sito za pronalaženje primbrojeva, Gaussov algoritam za rješavanje sustava linearnih jednadžbi, i slično.

Kako je uočio Menger: „Strukture podataka je malo teže definirati. Struktura podataka je skupina varijabli u nekom programu zajedno s vezama između tih varijabli. Strukture podataka su stvorene s razlogom kako bi omogućile lakše manipuliranje podatcima, odnosno efikasnije izvršenje operacija nad podatcima (upisivanje, promjena, čitanje, traženje po nekom kriteriju...)“ (Menger, 2014). Postoje različite strukture podataka (polje, lista, stog, red, graf,...), ali u ovom radu ćemo se fokusirati na listu.

2.2 Lista

Prema riječima Krešimira Kumeričkog: „Liste su središnji objekti programiranja u Pythonu. Srodne su poljima (array) iz standardnih programskih jezika, ali imaju bitno više svojstava. Elementi lista mogu biti praktički bilo koji objekti.“ (Kumerički, 2018). Liste koristimo u programiranju za spremanje bilo kojeg tipa podatka bilo to numerički, simbolički ili još jedna lista čak i prazan skup (*NULL*). Elementi liste mogu se zbrajati, oduzimati, množiti i dijeliti. Osim manipuliranja elementima liste manipulirati se može i samom listom kroz tzv. metode objekata. Metode objekata su skup naredbi koji nam služi za upravljanje svojstvima liste. Bilo dodavanja novog elementa (*.append*), oduzimanja (*.remove*), kopiranja (*.copy*), itd. Korisnost i sama kompleksnost lista dolazi do izražaja kada elemente neke liste čine druge liste. Ukoliko imamo listu kojoj je svaki element druga lista onda govorimo o matrici.

Matrica je zapravo dvodimenzionalni niz gdje se svaki element sastoji od podelemenata. Isto tako se može dogoditi da su elementi unutar liste također liste čiji su elementi liste. Onda govorimo o trodimenzionalnom nizu koji se koristi u grafičkim prikazima prostora gdje glavna lista predstavlja os X, podlista os Y i podpodlista os Z. Zatim kombinacijom indeksnih vrijednosti elemenata tih lista možemo prikazati bilo koje mjesto u trodimenzionalnom prostoru unutar programskog koda.

U ovom radu ćemo promatrati jednodimenzionalnu listu *li* čiji su elementi svi prirodni brojevi uključujući nulu.

3. ALGORITMI SORTIRANJA

Algoritmi za sortiranje služe kako bi uredili neuređen skup podataka, to mogu biti brojevi, slova ili znakovi. Kao kod bilo kojeg problema i ovdje postoji više točnih odgovora od kojih možemo birati, nisu svi jednako korisni, ovisno što želimo postići. Svaki algoritam je jedinstven i poseban na svoj način pa se zato različiti algoritmi za sortiranje koriste za različite ciljeve.

Bubble sort je daleko najlakši algoritam sortiranja sa kojim se može početi kako bi se upoznalo sa samim principom rada sortiranja i pretraživanja podataka. Budući da je on najjednostavniji za korisnika najlakše je objasniti princip sortiranja početniku. Zato jer je najjednostavniji za programera taj algoritam je „najteži“ za računalo i ne koristi se previše u praksi. Ima puno korisnijih algoritama obzirom na količinu podataka, brzinu sortiranja koju moramo zadovoljavati i samu kompleksnost.

Kod sofisticiranijih i kompliciranijih aplikacija koja koristi različite tipove podataka, koju koristi puno ljudi te je bitna brzina kojom krajnji korisnici dobiju svoje podatke koriste se „kompliciraniji“ algoritmi za sortiranje. Pod pojmom kompliciraniji misli se na mogućnost objašnjavanja tog algoritma većem broju ljudi. Algoritmi kao što su Shell sort, Quick sort, rekurzivni Quick sort su kompliciraniji za ispravno napisati, ali puno brže i efikasnije dolaze do krajnjeg rezultata.

R. Menger, 2014 je u svojoj knjizi podijelio algoritme za sortiranje u 4 kategorije:

- sortiranje zamjenom elemenata – obuhvaća sortiranje izborom najmanjeg elementa (engl. selection sort) i sortiranje zamjenom susjednih elemenata (engl. bubble sort).
- sortiranje umetanjem – postoji jednostavno sortiranje umetanjem (engl. insertion sort) i višestruko sortiranjem umetanjem (engl. shell sort).
- rekurzivni algoritmi za sortiranje – najbrži rekurzivni algoritam je algoritam brzog sortiranja (engl. quick sort) i algoritam sortiranja s pomoću hrpe (engl. heap sort).
- sortiranje pomoću binarnih stabla – algoritam sortiranja s pomoću hrpe (engl. heap sort).

3.1 Algoritmi pretraživanja kroz povijest

Uzimamo kao činjenicu stupanj razvijenosti Googlea i drugih tražilica. Napredni sistemi za kategoriziranje i rangiranje informacija, personalizirane rezultate pretraživanja nekada nisu bili toliko razvijeni kao u današnje vrijeme. Ukoliko nismo znali točno ime stranice koju želimo posjetiti ne bismo ju mogli pronaći. Rezultati pretraživanja su vrlo često bili zatrpani sa tzv. *spam* stranicama i novim poslodavcima su trebali mjeseci da indexiraju svoje podatke.

Prvi pretraživač[1] nazvan Archie je osmišljen 1990. od strane studenata Fakulteta računalnih znanosti Sveučilišta McGill kako bi se fakultet mogao povezati na internet. Omogućavao je korisnicima da pretražuju po internetu koristeći *FTP* (File Transfer Protocol) te je indeksirao datoteke za preuzimanje. Nakon Archie-a osmišljeno je na stotine algoritama za pretraživanje interneta od kojih je bitnije naglasiti: Google, Bing i Yahoo

Yahoo je prvi put korišten 1995.godine. Tehničari smatraju da je to jedan od temelja modernih internet pretraživača. Originalno, pretraživač je bio kolekcija „dobrih" web stranica. Koje su „dobre" stranice određivao je Yahoo tim. Nakon toga nadodan je opis stranice uz URL kako bi korisnici vidjeli što se nalazi na stranici bez da je posjete. To su bila vremena *dial-up* povezivanja na internet pa je opis uz URL uštedio puno vremena.

Danas ne možemo zamisliti internet bez Google tražilice. Google je pokrenut 1998. godine redefinirajući algoritam za pretraživanje interneta. Tajna Googlovog uspjeha je tzv. *backlinks.* Google nije rangirao stranice samo po popularnosti ili posjećenosti nego po spominjanju na drugim web stranicama. Jednostavnije rečeno web stranice bi rasle u popularnosti ako su spomenute na nekoj drugoj web stranici. Isto tako je uzeta u obzir provjerenost informacija na spomenutim web stranicama te njihova povezanost. Implementacijom tog jednostavnog načela *spam* stranice su nestale, a u isto vrijeme Google je stvorio ogromnu isprepletenu mrežu koja se mogla posjetit bilo gdje na planetu. Trenutno Googlove promjene u algoritmu pretraživanja diktiraju najbolji pristup optimizaciji web pretraživača.

Bing se pojavio na tržištu 2009. godine. Ljudi u šali govore o Bing-u (kada ga uspoređuju s Googlom) ali Bing posjeduje 14% pretraživanja u SAD-u. Osim *backlinks* Bing ujedno koristi i povezane prijedloge za pretraživanje. Što to znači za web pretraživače? Znači da pravilno

[1] Pretraživač - specijalizirano mrežno mjesto čija je glavna funkcija pomoć u pronalaženju informacija pohranjenih na drugim mrežnim mjestima (domenama)

iskorištene ključne riječi na web stranicama mogu rezultirati u velikim promjenama u rezultatima pretraživanja na webu.

Algoritmi pretraživanja imaju dugu povijest koja seže u rane dane računarstva. Evo kratkog pregleda nekih važnih prekretnica u povijesti algoritama pretraživanja:

- Linearno pretraživanje: algoritam linearnog pretraživanja najosnovniji je i najjednostavniji algoritam pretraživanja. Koristi se od ranih dana računalstva i još uvijek se koristi u mnogim aplikacijama danas. Linearno pretraživanje izvorno je implementirano pomoću hardvera kao što su bušene kartice i trake.
- Binarno pretraživanje: Binarno pretraživanje izumio je John Mauchly 1946. To je učinkovitiji algoritam od linearnog pretraživanja i koristi se za pretraživanje sortiranih nizova. Binarno pretraživanje izvorno se koristilo u ranim računalima kao što je UNIVAC I.
- Pretraživanje prvo u dubinu (DFS) i pretraživanje prvo u širinu (BFS): DFS i BFS su algoritmi za obilazak grafa koji su prvi put predstavljeni 1950-ih. Koriste se za istraživanje i pretraživanje grafikona i stabala[2]. DFS i BFS imaju mnoge primjene u računalnoj znanosti, uključujući umjetnu inteligenciju, rudarenje podataka i mrežnu analizu.
- A* pretraživanje: A* pretraživanje uveli su Peter Hart, Nils Nilsson i Bertram Raphael 1968. To je heuristički algoritam pretraživanja koji se koristi za pronalaženje najkraćeg puta između dva čvora u grafu ili mreži. A* Pretraživanje se široko koristi u robotici, programiranju igara i drugim aplikacijama.
- Web tražilice: Razvoj web tražilica u 1990-ima revolucionirao je način na koji ljudi traže informacije na internetu. Tražilice kao što je Google koriste složene algoritme za pretraživanje i indeksiranje ogromne količine informacija dostupnih na webu, čineći ih dostupnima korisnicima diljem svijeta.

Algoritmi pretraživanja nastavljaju se razvijati i poboljšavati, potaknuti napretkom tehnologije i sve većom količinom podataka koje je potrebno pretraživati i analizirati.

[2] Stablo - hijerarhijska struktura podataka koja se sastoji od čvorova koji su povezani granama

3.2 VREMENSKO TRAJANJE ALGORITMA

U radu s algoritmima postoje bolji i lošiji algoritmi. Pitanje je kako znamo koji je bolji, a koji lošiji? Za to postoje dva mjerila: vremensko trajanje i prostorna složenost. Vremensko trajanje je vrijeme koje je potrebno algoritmu da se u potpunosti izvrši. Na to utječe količina i kompleksnost instrukcija unutar algoritma, kao što su while i for petlje, grananje, i slično. Vremensko trajanje algoritma se računa tako da se uzme broj instrukcija u programu koje su bile aktivne te se pomnože sa konačnim vremenom za koje je računalo izvršilo taj algoritam.

Pod prostornom složenosti se misli na količinu memorijskog prostora koje algoritam mora prisvojiti kako bi se uspio izvršiti do kraja.

Prema Anany Levitinu postoje tri slučaja učinkovitosti:

- Najgora učinkovitost - najgori slučaj ulaznih podataka (svaki od članova je na krivom mjestu, trajanje algoritma je najdulje)
- Najbolja učinkovitost - najbolji slučaj ulaznih podataka (svaki od članova je na svojem mjestu, trajanje algoritma je najkraće)
- Prosječna učinkovitost - nasumični ulazni podaci (najčešće se događa u stvarnosti)

Kada se mjeri vremensko trajanje algoritma uvijek se uzima najgora učinkovitost, tj. najgore sortiran redoslijed ulaznih podataka. Tako se pripremamo za najgori mogući ishod i znamo što očekivati kada se najgori slučaj dogodi u stvarnosti, svi ostali slučajevi mogu samo biti bolji.

4. ALGORITMI PRETRAŽIVANJA

Algoritmi pretraživanja su algoritmi koji se koriste za pronalaženje određene vrijednosti ili stavke unutar sortirane liste podataka. Ovisno o vrsti načina pretraživanja, algoritmi se općenito klasificiraju u dvije kategorije:

- Sekvencijalno pretraživanje (Sequential Search). U ovom algoritmu lista ili niz se prelazi sekvencijalno i provjerava se svaki element kako pokazivač dođe do njegovog mjesta.
- Intervalno pretraživanje (Interval Search). Ovi su algoritmi posebno dizajnirani za pretraživanje velike količine sortiranih struktura podataka. Ova vrsta algoritama pretraživanja puno je učinkovitija od linearnog pretraživanja jer ciljaju središte sortirane liste u kojoj traže određeni podatak i dijele prostor pretraživanja na pola.

4.1 Algoritmi sekvencijalnog pretraživanja

Algoritmi sekvencijalnog pretraživanja su najjednostavniji algoritmi za početnike, te samim time su prvi algoritmi na kojima se uči principi sortiranja i pretraživanja. Kada su podatci pohranjeni na skupu kao što je lista, kažemo da imaju sekvencijalni ili linearan odnos. Svaki je podatak pohranjen u odnosu na svojeg prethodnika i svojeg sljedbenika. U Python listama ti relativni položaji su vrijednosti indeksa pojedinačnih stavki, odnosno podatka koji se nalazi na toj indeksnoj vrijednosti. Budući da su ove vrijednosti indeksa poredane moguće ih je posjećivati redom. Ovaj proces dovodi do prve tehnike pretraživanja, sekvencijalnog pretraživanja.

4.2 Linearno pretraživanje

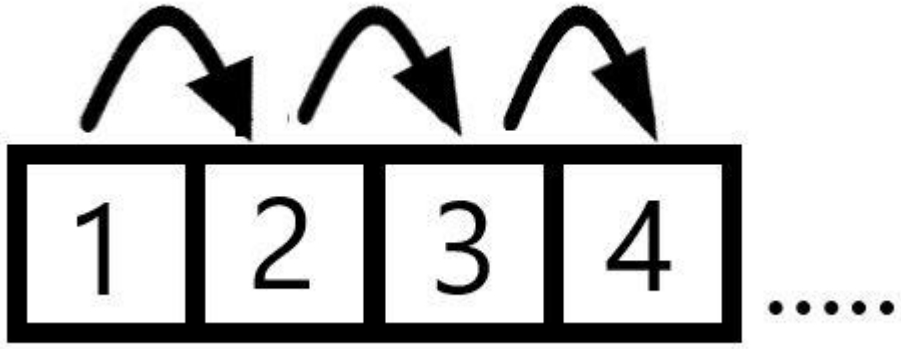
Linearno ili sekvencijalno pretraživanje funkcionira na vrlo jednostavnom principu. Unutar dane liste (u ovom slučaju lista se sastoji od cijelih brojeva) pretražuje unaprijed određenu vrijednost tako što uspoređuje svaki od članova dane lista. Počinje na prvom mjestu, tj. pokazivač koji nam služi za „hodanje“ po listi stavljamo na indeks koji ima vrijednost 0. Nakon što provjeri da li je tražena vrijednost na trenutnom indeksnom mjestu postoje dva slučaja. Ukoliko je vrijednost ista vraća broj indeksa na kojem se nalazi tražena vrijednost. Ukoliko tražena vrijednost nije pronađena na trenutnom indeksu pokazivač prelazi na sljedeći element te se ponavlja usporedba dok se ne pronađe tražena vrijednost. Ako dana lista ne sadrži vrijednost koju zadamo, program vraća -1.

Prvo definiramo listu (kod linearnog pretraživanja ne mora biti sortirana) sa imenom lista te joj odmah definiramo elemente. Nakon toga u varijablu *x* spremamo vrijednost koju tražimo u listi. U ovom slučaju koristimo funkciju *rand()* koja nasumično izabire jedan broj. Sami algoritam definiramo ispred svega te ga nazivamo *search*. Unutar *search* odlomka je zapravo naš algoritam pretraživanja koji koristi linearno pretraživanje kroz petlje i grananja. Nakon što završi algoritam, ispisujemo što je taj algoritam vratio, tj. njegovu vrijednost koja je spremljena u varijablu i, odnosno vraća -1 ako element ne postoji u listi. Isti primjer napisan Phyton jezikom bi izgledao ovako:

```
import time
import random

def search(lista, n, x):
    for i in range(0, n):
        if (lista[i] == x):
            return i
    return -1

start = time.time()
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x = random.randint(1, 10)
n = len(lista)

rezultat = search(lista, n, x)
print("Vrijednost koju tražimo je: ", x)
if (rezultat == -1):
    print("Vrijednost se ne nalazi u listi.")
else:
    print("Vriejdnost se nalazi na indeksu broj:", rezultat)
end = time.time()
print("Vremensko trajanje ovog algoritma je: ", end - start)
```

Slika 1 Linearno pretraživanje u Phytonu
Izvor: Izrada autora

Radi lakšeg razumijevanja imamo vizualni primjer (Slika 2) kako algoritam pretražuje brojeve linearno dok ne dođe do tražene vrijednosti



Slika 2: Linearno pretraživanje

4.2.1 Vremenska kompleksnost linearnog pretraživanja

Internetski blog Geeks for Geeks navodi kako je vremenska kompleksnost ovog algoritma *O(n)*. Gdje n označava duljinu liste. Ova vrsta pretraživanja rijetko se koristi u praksi jer drugi algoritmi pretraživanja (kao što su algoritam binarnog pretraživanja i hash tablice) omogućuju znatno brže pretraživanje u usporedbi s linearnim pretraživanjem.

Korištenjem funkcije *time()* možemo točno u sekundama vidjeti koliko je potrebno programu da se izvrši.

4.3 Algoritmi intervalnog pretraživanja

Algoritmi intervalnog pretraživanja rade na principu „*divide and conquer*“ (podijeli pa vladaj). Intervalno pretraživanje je jedino moguće kod sortiranih podataka, ukoliko podatci nisu sortirani rezultat koji ćemo dobiti neće biti valjan. Dobar algoritam intervalnog pretraživanja za početnike jest Binarno pretraživanje. Binarno pretraživanje uzima sortiranu listu podataka te gleda prvenstveno vrijednost njenog srednjeg elementa, zatim ga uspoređuje s unaprijed zdanom vrijednošću koja se traži. Nakon uspoređivanja algoritam odlučuje koju polovicu sortirane liste će koristiti jer se u toj polovici nalazi tražena vrijednost, a drugu polovicu zanemaruje. U polovici u kojoj se nalazi tražena vrijednost ponovo uzima vrijednost koja se nalazi na sredini te se postupak ponavlja dok ne ostane samo jedna vrijednost, odnosno tražena.

4.3.1 Primjer binarnog pretraživanja

Uzmemo li našu sortiranu listu li[1, 2, 3, 4,..,10] (radi lakšeg pojašnjena uzmimo da je 10 zadnja vrijednost u listi) te ju ubacimo u algoritam, dodijelimo svakoj vrijednosti njenu indeksnu vrijednost dobit ćemo nešto slično ovome:

Slika 3.Prvi korak binary search

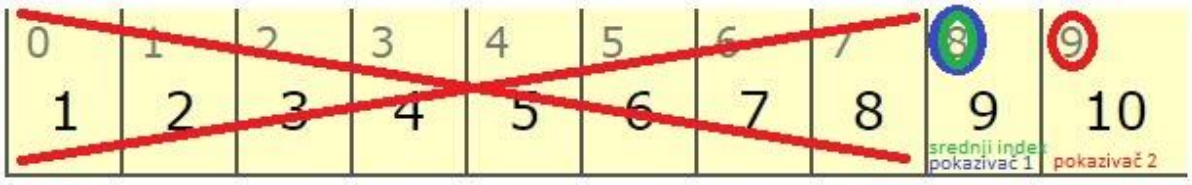
Pretpostavimo da je naša *rand()* funkcija nasumično odabrala vrijednost koju traži da je jednaka broju 9. Prvo što algoritam radi jest postavlja dva pokazivača na prvo i zadnje mjesto u listi (točnije jedan pokazivač postavlja na mjesto s najmanjim indeksom, a drugi pokazivač postavlja na mjesto s najvećim indeksom) i jedan na sredinu, te uzima njihove vrijednosti.

Slika 4. Binary pokazivači

Nakon što ima tražene tri vrijednosti uspoređuje srednju s traženom i gleda u kojoj se polovici nalazi tražena vrijednost, odnosno dali je tražena vrijednost veća ili manja od srednje. U našem slučaju je veća jer je srednja vrijednost 5, a tražena je 9. Tako da algoritam zaključuje da se veći brojevi nalaze s desne strane liste i algoritam odbacuje lijevu stranu liste, odnosno on vidi:

Slika 5. Binary pokazivači 2.korak

Zatim pomiče pokazivače na najmanju, najveću i srednju indeksnu vrijednost (kao što vidimo na slici 5) te ponovo uspoređuje vrijednost na kojoj se nalazi srednji indeks s traženom vrijednošću. U našem slučaju ponovno zanemaruje lijevu stranu liste jer tražena vrijednost je 9, a srednja je 8 tako da tražena vrijednost je veća od srednje, te zaključuje da se nalazi na desnoj strani od srednjeg indexa pa dobijemo slijedeće:



Slika 6. Binary kraj

Ovdje dolazimo do vrlo bitnog momenta, a to je da pokazivač i srednji index pokazuju na isti index. To za naš algoritam binarnog pretraživanja znači da se tražena vrijednost nalazi na tom indexnom mjestu. Nakon što se dođe do ovog koraka uzimamo vrijednost od dobivenog indexa te je vraćamo krajnjem korisniku.

Primjer obrađen gore pisan Phyton jezikom bi izgledao ovako:

```
import time
import random
def binarySearch(li, l, r, x):
    while l <= r:
        mid = l + (r - l)
        if li[mid] == x:
            return mid
        elif li[mid] < x:
            l = mid + 1
        else:
            r = mid - 1
    return -1

start = time.time()
li = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x = random.randint(1, 10)

rezultat = binarySearch(li, 0, len(li) - 1, x)
print("Vrijednost koju tražimo je: ", x)
if rezultat == -1:
    print("Element se ne nalazi u listi.")
else:
    print("Element se nalazi na indeksu broj: ", rezultat)

end = time.time()
print("Vremensko trajanje opvog algoritma je: ", end - start)
```

Slika 7 Binarno pretraživanje u Phytonu
Izvor: Izrada autora

4.3.2 Jump search algoritam

Kako i kod binarnog pretraživanja *jump search* algoritam koristi unaprijed sortiranu listu podataka. Funkcionira na jednostavnom principu „preskakanja“ elemenata u listi, po čemu je i dobio naziv *jump search*. Uzmimo našu sortiranu listu li[1, 2, 3, 4, 5, 6,..., 10^n] kao primjer radi lakšeg pojašnjena principa rada jump search algoritma. Neka je unaprijed zadana vrijednost koju algoritam traži broj: 17. Osim što smo mu zadali vrijednost koju traži moramo mu zadati i interval koji „preskače“, neka je to 5. Znači naš algoritam će prvo „skočiti“ za 5 mjesta u našoj listi te usporediti vrijednost koja se nalazi na tom mjestu. U našem slučaju to je element s indeksom 4, odnosno broj 5. zatim uspoređuje da li je broj

manji od tražene vrijednosti, ako nije ponovo „skače“ na sljedeći, odnosno element s indeksom 9, tj. broj 10. Tako se ponavlja dok ne nađe broj koji je veći od traženog elementa i onda jednostavnim linearnim pretraživanjem prođe kroz zadnjih nekoliko elemenata (u našem slučaju 5) dok ne nađe traženu vrijednost.

1. Korak: Skoči na indeks 4, usporedi
2. Korak: Skoči na indeks 9, usporedi
3. Korak: Skoči na indeks 14, usporedi
4. Korak: Skoči na indeks 19, usporedi
5. Korak: Nađena je vrijednost veća od tražene vrijednosti
6. Korak: Linearno pretraži vrijednosti s indeksima 14, 15, 16, 17, 18

Kod *jump search* algoritma se postavlja pitanje kojeg nemamo u drugim algoritmima pretraživanja: *Koji je optimalni broj elemenata koji bi trebao preskočiti*? U najgorem slučaju, moramo napraviti n/m skokova (gdje je n broj elemenata u listi, a m je broj elemenata koji preskačemo), a ako je zadnja provjerena vrijednost veća od elementa koji se traži, izvodimo m-1 usporedbi uz pomoć linearnog pretraživanja. Stoga će ukupni broj usporedbi u najgorem slučaju biti ((n/m) + m-1). Vrijednost funkcije ((n/m) + m-1) bit će minimalna kada je m = √n. Stoga je najbolja veličina koraka m = √n, što znači da je kompleksnost ovog algoritma O(√n).U odnosu na linearno i binarno pretraživanje *jump* algoritam za pretraživanje je bolji od linearnog, ali je binarni bolji od *jump* algoritma za pretraživanje. Treba naglasiti da *jump search* ima prednost u odnosu na binarni zato što se „vraća“ samo jednom. Recimo da tražimo vrlo mali element, binarno pretraživanje bi imalo isto koraka kao da tražimo veliki element dok bi *jump search* algoritam odmah u prvom koraku znao da je traženi element unutar prvih nekoliko elemenata liste. U sustavima gdje je skupo puno puta pretraživati koristimo jump pretraživanje.

Isti primjer pisan u phyton kodu:

```
import math
import random
import time

def jumpSearch(li, x, n):
    korak = math.sqrt(n)

    prev = 0
    while li[int(min (korak, n) -1)] < x:
        prev = korak
        korak += math.sqrt(n)
        if prev >= n:
            return -1
    while li[int(prev)] < x:
        prev += 1
        if prev == min (korak, n):
            return -1
    if li[int(prev)] == x:
        return prev
    return -1

start = time.time()
li = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x = random.randint(1, 10)
n = len(li)

rezultat = jumpSearch(li, x, n)
print("Vrijednost koju tražimo je: ", x)

if rezultat == -1:
    print("Element se ne nalazi u listi.")
else:
    print("Element se nalazi na indeksu broj: ", rezultat)

end = time.time()
print("Vremensko trajanje ovog algoritma je: ", end - start)
```

Slika 8 Jump pretraživanje u phytonu
Izvor: Izrada autora

4.3.3 Interpolation search algoritam

Interpolacijsko pretraživanje je unaprijeđena verzija binarnog pretraživanja. Da bi interpolacijsko pretraživanje moglo raditi, podaci unutar dane liste moraju biti ravnomjerno raspoređeni. Binarno pretraživanje uvijek dijeli listu na dva jednaka dijela dok interpolacijsko koristi posebnu formulu kako bi se odredilo u kojem se kraju liste traženi element nalazi. Osnovni princip formule jest: Što je dobiveni broj veći, to je tražena vrijednost bliže kraju liste.

$$POZ = I_0 + \left[\frac{(x - li[I_0]) * (hi - I_0)}{li[hi] - li[I_0]}\right]$$

POZ - pozicija koju odabiremo
li[] = dana lista koja se pretražuje
x = vrijednost koju tražimo
hi = vrijednost zadnjeg indexa u listi li[]
l_0 = vrijednost prvog indexa u listi li[]

Niže je opisano kako funkcionira interpolacijsko pretraživanje po koracima.

1. Koristeći gore napisanu formulu dobije se vrijednost POZ.
2. Ako se ta vrijednost podudara s traženom algoritam završava, te smo dobili indeks na kojem se nalazi tražena vrijednost.
3. Ako je dobivena vrijednost POZ manja od tražene vrijednosti ponavljamo 2. korak s lijevom stranom liste (gdje se nalaze manje vrijednosti), odnosno ako je dobivena vrijednost POZ veća od tražene vrijednosti ponavljamo 2. korak s desnom stranom liste (gdje se nalaze veće vrijednosti).
4. Zatim ponavljamo sve dok vrijednost POZ ne odgovara traženoj vrijednosti ili dok se lista ne može podijeliti na manje, a u tom slučaju znači da se tražena vrijednost ne nalazi u danoj listi.

Internetski članak *Interpolation search* objavljen na web stranici Geeks for Geeks navodi kako je vremenska kompleksnost linearnog pretraživanja O(n), *jum search* algoritmu treba O(√n) vremena, binarnom O(log(n)) vremena. Kako interpolacijsko pretraživanje nema unaprijed određenu vrijednost gdje počinje, prosječna kompleksnost interpolacijskog pretraživanja je $O(\log(\log n))$

Isti program pisan Phyton jezikom (koristeći listu li[1, 2, 3, 4, 5, 6,..., 10^n]) izgleda ovako:

```
import time
import random

def interpolationSearch(li, lo, hi, x):
    if (lo <= hi and x >= li[lo] and x <= li[hi]):
        poz = lo +((hi - lo) // (li[hi] - li[lo]) * (x - li[lo]))
        if li[poz] == x:
            return poz
        if li[poz] < x:
            return interpolationSearch(li, poz + 1, hi, x)
        if li[poz] > x:
            return interpolationSearch(li, lo, poz -1, x)
    return -1

start = time.time()
li = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x = random.randint(1, 10)
n = len(li)

rezultat = interpolationSearch(li, 0, n - 1, x)

if rezultat == -1:
    print("Element se ne nalazi u listi.")
else:
    print("Element se nalazi na indeksu broj: ", rezultat)

end = time.time()
print("Vremensko trajanje opvog algoritma je: ", end - start)
```

Slika 9 Algoritam pretraživanja interpolacijom pisan u Phytonu
Izvor: Izrada autora

4.3.4 Eksponencijalno pretraživanje

Eksponencijalno pretraživanje funkcionira na sličnom principu kao i *jump search* algoritam jer u kasnijim koracima koristi pomoć drugog algoritma pretraživanja, binarnog ili linearnog u ovom slučaju. Postoje dva osnovna koraka:

1. Eksponencijalno se pronalazi podlista u kojoj se nalazi tražena vrijednost
2. Binarno ili linearno se pretražuje dobivena podlista u kojoj se nalazi vrijednost

Isto kao i svi algoritmi do sad dobivena lista mora biti unaprijed sortirana kako bi algoritam ispravno radio. Problem na koji nailazi je kako odrediti podlistu u kojoj se nalazi tražena vrijednost? Samo ime nam govori, eksponencijalno, kako algoritam radi. Za početak uzima se prvi element te njega smatra kao podlistu u kojoj se nalazi vrijednost i uspoređuje zadnji element u podlisti, u ovom slučaju i jedini. Ukoliko je tražena vrijednost veća, proširuje veličinu podliste na prva dva elementa, ponovo uspoređuje sa zadnjim, odnosno drugim u ovom slučaju, pa ako je tražena vrijednost veća proširuje podlistu na prva 4 elementa, pa nakon toga na prvih 8 i tako dalje dok tražena vrijednost nije manja od zadnjeg elementa trenutne podliste. Nakon što nađe manji element od tražene vrijednosti zna da se traženi element nalazi negdje između prijašnjeg indeksa koji je eksponencirao i onoga s kojim trenutno uspoređuje. Dobivenu podlistu binarno ili linearno pretražuje kako bi dobio indeks na kojem se nalazi zadana vrijednost. Vremenska složenost ovog algoritma je O(log n)

Pretraživanje naše liste, li[1,2, 3, 4,..., 10^n] koja se sastoji od n prirodnih brojeva, korištenjem eksponencijalnog pretraživanje koje poziva binarno izgleda ovako:

```
import time
import random

def binarySearch(li, l, r, x):
    if r >= l:
        mid = l + (r - l) // 2
        if li[mid] == x:
            return mid
        if li[mid] > x:
            return binarySearch(li, l, mid - 1, x)
        return binarySearch(li, mid + 1, r, x)

def exponentialSearch (li, n, x):
    if li[0] == x:
        return 0
    i = 1
    while i < n and li[i] <= x:
        i = i * 2
    return binarySearch(li, i // 2, min(i, n - 1), x)

start = time.time()
li = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x = random.randint(1, 10)
n = len(li)

rezultat = exponentialSearch(li, n, x)

if rezultat == -1:
    print("Element se ne nalazi u listi.")
else:
    print("Element se nalazi na indeksu broj: ", rezultat)

end = time.time()
print("Vremensko trajanje opvog algoritma je: ", end - start)
```

Slika 10 Algoritam eksponencijalnog pretraživanja pisan u Phytonu
Izvor: Izrada autora

4.3.5 Fibonacci search algoritam

Fibonaccievo pretraživanje funkcionira na sličnom principu kao i binarno pretraživanje. Točnije radi na principu podijeli i vladaj (divide and conquer), može raditi samo sa unaprijed sortiranim listama podataka, te ima vremensku kompleksnost O(log n) gdje je n broj elemenata u listi. Bitno je naglasiti i razlike: Fibonaccievo pretraživanje ne dijeli listu na dvije manje liste s jednakim brojem elemenata, binarno pretraživanje se koristi operatorom dijeljenje kako bi podijelilo listu na manje, Fibonaccievo pretraživanje koristi operatore plus i minus kako bi podijelio listu na manje podliste. Operator dijeljenja (koji se koristi u binarnom pretraživanju) može predstavljati problem nekim procesorima zbog kompleksnosti operatora. Fibonaccievo pretraživanje uspoređuje relativno bliske elemente u listi, samim time ako je lista prevelika i ne stane u RAM memoriju pretraživanje neće biti uspješno.

Da bi mogli objasniti Fibonaccievo pretraživanje prvo moramo opisati što je Fibonacciev niz. Kako navodi autorica Suzana Dobrić prva dva člana Fibonaccijeva niza su 1 i 1, a svaki sljedeći član dobije se tako da se zbroje prethodna dva: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610… Prethodna rečenica izražena matematičkom formulom izgleda ovako

$F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1$, gdje je n bilo koji broj u Fibbonacievom nizu.

Način na koji funkcionira Fibonacciev niz je sljedeći: prvo se pronađe najmanji broj Fibonaccievog niza koji je veći od broja elemenata u listi. Zatim se uzmu dva prethodna broja Fibonaccievog niza (F(n-1) i F(n-2)). Kada imamo F(n-2) indeks u listi njen element uspoređujemo s traženom vrijednošću. Ako je to tražena vrijednost, našli smo indeks na kojem se nalazi. Ako je tražena vrijednost manja pomičemo elemente u listi na prethodna dva elementa Fibonaccievog niza, time eliminiramo dvije trećine s kraja niza. Ako je tražena vrijednost veća pomičemo elemente u listi na prethodni element Fibonaccievog niza te stavljamo pomak na indeksnu vrijednost, s time eliminiramo prednju trećinu liste. Nakon tih koraka ostaje samo još jedna usporedba, a to je usporediti traženi element sa drugim elementom Fibonaccievog niza tj. F(n-1). Ukoliko se dođe do zadnjeg koraka najčešće je to i traženi indeks.

Primjer Fibonaccievog pretraživanja

Uzmimo našu listu *li*[1,2,3,...,10^n] (prvih 9 elemenata radi jednostavnosti) te ju prikažimo vizualno:

INDEKS (i)	0	1	2	3	4	5	6	7	8	9
ELEMENT	1	2	3	4	5	6	7	8	9	10

Slika 11. Fibbonachi - 1

Nakon što imamo našu listu uzimamo Fibonacciev niz te određujemo najmanji element koji je veći od broja elemenata u listi (1, 1, 2, 3, 5, 8, 13,...), odnosno broj 13. Sad kad imamo F(n)=13 možemo izračunati prethodna dva elementa koji su F(n-1) = 8 i F(n-2) = 5. Za primjer uzmimo da tražimo broj

10, odnosno element na indeksu 9. Budući da je vrijednost pomaka indeks i svi indeksi uključujući njega i nakon njega su eliminirani, jedino ima smisla dodati joj nešto. Budući da F(n-2) označava otprilike jednu trećinu našeg niza, kao i indeksi koje označava, sigurno su valjani, možemo dodati F(n-2) za pomak i provjeriti element na indeksu i = min (pomak + F(n-2), n). Korak po korak unutar tablice, nakon što se nađe najmanji Fibonacciev broj izgleda ovako:

F(n-2)	F(n-1)	F(n)	Pomak	i = min (pomak + F(n-2), n)	Li[i]	Korak
5	8	13	0	5	6	Pomak dva elementa nazad
3	5	8	5	8	9	Pomak dva elementa nazad
2	3	5	9	9	10	Vrati pripadajući ideks

Slika 12. Fibbonachi - 2

Vremenska kompleksnost Fibonaccievog pretraživanja je najgora kada se traženi element nalazi u one 2/3 liste i uvijek bude unutar te 2/3. Nakon prvog pretraživanja je (2/3)*n, nakon drugog je (4/9)n, nakon trećeg (8/27)*n, i tako dalje.... (Geeks for Geeks, 2022). Formula za kompleksnost Fibonaccievog niza je: O(log n) gdje je n broj elemenata u nizu.

Fibbonacievo pretraživanje napisano Phyton kodom :

```
import time
from bisect
from bisect import bisect_left
import time
import random

def fibMonaccianSearch(li, x, n):
    fibMMm2 = 0
    fibMMm1 = 1
    fibM = fibMMm2 + fibMMm1

    while (fibM < n):
        fibMMm2 = fibMMm1
        fibMMm1 = fibM
        fibM = fibMMm2 + fibMMm1

    pomak = -1

    while (fibM > 1):

        i = min(pomak+fibMMm2, n-1)

        if (li[i] < x):
            fibM = fibMMm1
            fibMMm1 = fibMMm2
            fibMMm2 = fibM - fibMMm1
            pomak = i

        elif (li[i] > x):
            fibM = fibMMm2
            fibMMm1 = fibMMm1 - fibMMm2
            fibMMm2 = fibM - fibMMm1

        else:
            return i

    if(fibMMm1 and li[n-1] == x):
        return n-1

    return -1
```

```
start = time.time()
li = list(range(1, 101))
n = len(li)
x = random.randint(1, 100)
index = fibMonaccianSearch(li, x, n)
print("Tražena vrijednost je: ", x)
if index>=0:
  print("Vrijednost se nalazi na indexexu:",index)
else:
  print(x,"Vrijesnot nije u listi")
end = time.time()
print("Vremensko trajanje opvog algoritma je: ", end - start)
```

Slika 13 Algoritam Fibonaccijevog pretraživanja u Phytonu
Izvor: Izrada autora

4.3.6 Hash Table Search Algorithm

Algoritam pretraživanja hash tablica je algoritam koji se koristi za brzo dohvaćanje podataka iz velikog skupa podataka preslikavanjem podataka u par ključ-vrijednost (*eng. „key-value"*). U ovom algoritmu, hash funkcija se koristi za generiranje jedinstvene vrijednosti indeksa za svaki dio podataka, a ovaj indeks se koristi za pohranjivanje i dohvaćanje podataka iz tablice.

Kada se izvrši pretraživanje, hash funkcija se primjenjuje na ključnu vrijednost koja se traži, a rezultirajući indeks koristi se za lociranje podataka u skupu podataka. Ako su podaci u danom skupu, mogu se brzo dohvatiti pomoću indeksa.

Učinkovitost algoritma pretraživanja hash tablice ovisi o kvaliteti hash funkcije koja se koristi, kao i o veličini hash tablice. Ako hash funkcija proizvodi previše kolizija (gdje se više ključeva preslikava na isti indeks), pretraživanje može biti sporije zbog potrebe za rješavanjem kolizija. Osim toga, ako je hash tablica premala, pretraživanje također može biti sporije zbog veće šanse za kolizije.

Jedna od prednosti hash tablica je ta da pružaju izvođenje prosječnog slučaja u konstantnom vremenu za pretraživanje, umetanje i brisanje elemenata, pod pretpostavkom da hash funkcija ravnomjerno raspoređuje ključeve po nizu.

Međutim, može doći do sudara kada se različiti ključevi mapiraju na isti indeks, što rezultira sporijom izvedbom. Postoji nekoliko strategija za rukovanje kolizijama, uključujući ulančavanje (pohranjivanje povezanog popisa elemenata u svakom spremniku), otvoreno adresiranje (ispitivanje sljedećeg dostupnog utora) i rehashing (ponovna izgradnja hash tablice s većom veličinom ili drugom hash funkcijom) .

Hash tablice se naširoko koriste u računalnim znanostima i programiranju, a koriste se u mnogim aplikacijama kao što su predmemorija, tablice simbola, indeksiranje baza podataka i kriptografija. Neki programski jezici, kao što je Python, pružaju ugrađenu podršku za hash tablice putem svoje vrste podataka rječnika.

Primjer Hash Table Search algoritma u Pythonu:

```
# stvaranje „riječnika“ tj. dictionary-a
rijecnik = {'jabuka': 2, 'banana': 4, 'naranča': 1}
# ispis „riječnika“
print(rijecnik)
# pristupanje vrijednosti preko ključa
print(rijecnik ['banana'])
# dodavanje novog ključ-vrijednost para
rijecnik ['grožđe'] = 3
# ispis ažuriranog riječnika
print(rijecnik)
# polaženje kroz sve ključeve i vrijednosti u rječniku
for key, value in rijecnik.items():
    print(key, value)
```

Slika 14 Algoritam hash pretraživanja u Phytonu

Izvor: izrada autora

5. Usporedba algoritama pretraživanja

Kao što je već rečeno Python je široko korišten programski jezik s nizom korisnih biblioteka[3] za razvoj svakojakih aplikacija. U Pythonu je dostupno nekoliko algoritama pretraživanja, svaki sa svojim jedinstvenim prednostima i nedostacima. Usporedba ovih algoritama može pomoći u identificiranju najprikladnijeg algoritma za određeni problem pretraživanja, uzimajući u obzir veličinu skupa podataka, prirodu podataka i potrebno vrijeme pretraživanja.

U ovom ćemo djelu istražiti i usporediti neke od najpopularnijih algoritama pretraživanja dostupnih u Pythonu, poput linearnog pretraživanja, binarnog pretraživanja, hash tablice, eksponencijalnog i više. Analizirat ćemo izvedbu svakog algoritma na temelju vremena pretraživanja, složenosti prostora i jednostavnosti implementacije. Uspoređujući algoritme pretraživanja u Pythonu, možemo steći bolje razumijevanje njihovih prednosti i ograničenja te donijeti informirane odluke o tome koji algoritam koristiti za specifične probleme pretraživanja.

5.1 Linearno pretraživanje – usporedba

Linearno pretraživanje najjednostavniji algoritam pretraživanja koji prolazi svaku stavku u zbirci podataka kako bi pronašao ciljnu stavku. Ima vremensku složenost O(n), gdje je n broj elemenata u danom skupu podataka. Algoritam linearnog pretraživanja dobro funkcionira za male skupove podataka, ali može biti neučinkovit za velike količine podataka. Ostali algoritmi pretraživanja, kao što je binarno pretraživanje, hash tablica i interpolacijsko pretraživanje, općenito su brži od linearnog pretraživanja za veće skupove podataka. Samim time njegova iskoristivost osim objašnjavanja principa pretraživanja podataka je vrlo mala.

5.2 Binarno pretraživanje – usporedba

Binarno pretraživanje popularan je algoritam pretraživanja koji učinkovito radi za sortirane nizove. Ima vremensku složenost od O(log n), što ga čini bržim od linearnog pretraživanja za veće skupove podataka. Međutim, binarno pretraživanje zahtijeva da podaci budu sortirani, što može dodati dodatnu vremensku složenost za prethodnu obradu ili sortiranje podataka.

U usporedbi s Hash tablicom koje je podatkovna struktura i koja se može koristiti za učinkovito pretraživanje i dohvaćanje podataka. Ima prosječnu vremensku složenost od O(1) za operacije umetanja, brisanja i pretraživanja, što je brže od binarnog pretraživanja. Međutim, hash tablica može imati vremensku složenost O(n) u najgorem slučaju zbog rješavanja kolizije.

[3] Biblioteka - zbirka potprograma koji nude rješenja tematski povezanim problemima.

5.3 Jump search – usporedba

Jump search je algoritam pretraživanja koji se obično koristi za pretraživanje sortiranih nizova. Djeluje tako da listu podijeli na manje blokove i zatim izvrši linearno pretraživanje nad dobivrniom blokovima. Algoritam određuje veličinu bloka uzimajući kvadratni korijen veličine polja. Preskočno pretraživanje ima vremensku složenost O(√n), gdje je n broj stavki u nizu.

U usporedbi s drugim algoritmima pretraživanja, jump search općenito je brži od linearnog pretraživanja, ali sporiji od binarnog pretraživanja. Binarno pretraživanje ima vremensku složenost O(log n), što je brže od jump search za veće skupove podataka. Međutim, jump search algoritam je učinkovitiji od binarnog pretraživanja za manje skupove podataka, osobito kada podaci nisu ravnomjerno raspoređeni.

Ukratko, jump search koristan je algoritam za pretraživanje sortiranih nizova, osobito za manje skupove podataka. Binarno pretraživanje općenito je brže od jump search algoritma za veće skupove podataka. Hash tablica i interpolacijsko pretraživanje korisni su za pretraživanje ne sortiranih skupova podataka, odnosno jednoliko raspodijeljenih skupova podataka.

5.4 Interpolation search - usporedba

Interpolacijsko pretraživanje je algoritam pretraživanja koji procjenjuje položaj ciljane vrijednosti u listi na temelju njezine vrijednosti i distribucije jedinica u skupu. Posebno je koristan za pretraživanje jednolično raspoređenih skupova podataka. Algoritam koristi interpolacijsku formulu za procjenu položaja ciljne vrijednosti u zadanom skupu podataka, a zatim izvodi binarno pretraživanje kako bi pronašao ciljnu stavku. Pretraživanje interpolacijom ima vremensku složenost od O(log log n) za ravnomjerno raspoređene skupove podatak, a u najgorem mogućem slučaju vremenska kompleksnost mu je O(n).

U usporedbi s drugim algoritmima pretraživanja, interpolacijsko pretraživanje općenito je brže od linearnog pretraživanja i preskočnog pretraživanja, ali sporije od binarnog pretraživanja za veće skupove podataka. Linearno pretraživanje ima vremensku složenost O(n), gdje je n broj stavki u skupu, što ga čini ne učinkovitim za velike skupove podataka. Jump search je brže od linearnog pretraživanja, ali sporije od binarnog pretraživanja. Jump search ima vremensku složenost O(√n), gdje je n broj stavki u skupu.

5.5 Eksponential search – usporedba

Eksponencijalno pretraživanje je algoritam pretraživanja koji funkcionira tako da prvo pronađe raspon koji sadrži ciljni element, a zatim izvrši binarno pretraživanje unutar tog raspona kako bi se pronašao ciljni element. Djeluje tako da udvostručuje veličinu raspona pri svakoj iteraciji dok se ne pronađe ciljni

element ili dok raspon ne premaši veličinu skupa. Eksponencijalno pretraživanje ima vremensku složenost od O(log n) i za prosječni i za najgori scenarij.

U usporedbi s drugim algoritmima pretraživanja, eksponencijalno pretraživanje općenito je brže od linearnog pretraživanja, jump search-a i interpolacijskog pretraživanja za veće skupove podataka. Linearno pretraživanje ima vremensku složenost O(n), gdje je n broj stavki u zbirci, što ga čini neučinkovitim za velike skupove podataka. Jump search i pretraživanje interpolacijom brže su od linearnog pretraživanja, ali sporije od binarnog pretraživanja i eksponencijalnog pretraživanja. Jump search ima vremensku složenost od O(√n), gdje je n broj stavki u skupu, a interpolacijsko pretraživanje ima vremensku složenost od O(log log n) za jednoliko raspodijeljene skupove podataka i O(n) za najgori scenariji.

5.6 Fibonacci search – usporedba

Fibonaccijevo pretraživanje funkcionira dijeljenjem sortiranog niza u Fibonaccijeve brojeve i korištenjem Fibonaccijevih brojeva za određivanje pozicije sljedeće usporedbe. Djeluje slično binarnom pretraživanju, ali umjesto dijeljenja niza na pola, dijeli niz na dva dijela s veličinama proporcionalnima Fibonaccijevim brojevima. Fibonaccijevo pretraživanje ima vremensku složenost O(log n) za prosječni i najgori scenarij.

U usporedbi s drugim algoritmima pretraživanja, Fibonaccijevo pretraživanje općenito je brže od linearnog pretraživanja, jump search, interpolacijskog pretraživanja i eksponencijalnog pretraživanja za veće skupove podataka. Linearno pretraživanje nije učinkovito sa velike skupove podataka. Jump search i interpolacijsko pretraživanje brže su od linearnog pretraživanja, ali sporije od binarnog pretraživanja i Fibonaccijevog pretraživanja. Eksponencijalno pretraživanje, kao što je ranije spomenuto, općenito je brže od linearnog pretraživanja, preskočnog pretraživanja i interpolacijskog pretraživanja, ali Fibonaccijevo pretraživanje može biti brže za veće skupove podataka.

6. Algoritmi pretraživanja u praksi

Kao što je prije u radu navedeno algoritmi pretraživanja najviše se koriste u web tražilicama, ali imaju svoju primjenu u pretraživanju raznih zapisa podataka, bilo iz sigurnosnih ili analitčkih razloga. Na primjer u pravnim ustanovama kod identificiranja čovjeka po njegovoj slici ili otisku prsta koristi se algoritam za pretraživanje nad velikom bazom podataka. Isto tako može se koristiti za rješavanje problema kao što je problem trgovačkog putnika gdje se mora pronaći najkraći, ali i najefikasniji put između dane liste specifičnih odredišta. Rješenja takvih problema su korisna u logistici kod pronalaženja optimalne rute vozila.

Primjena pretraživanja se može kategorizirati u četiri glavne kategorije kada se koriste: eksplicitno pohranjene baze podataka, virtualni prostori pretraživanja, podstrukture dane strukture i konačno kvantna računala budućnosti.

6.1 Problem trgovačkog putnika (eng. Traveling salesman problem)

Navedeni problem je algoritamski problem koji ima zadatak pronaći najkraći put između skupa točaka i lokacija koje je potrebno posjetiti. U iskazu problema, točke su gradovi koje trgovački putnik može posjetiti. Cilj trgovačkog putnika je da troškovi putovanja i prijeđena udaljenost budu što je moguće niži. Naglasak ovdje je na optimizaciji (Whatls.com, 2022).

Problem trgovačkog putnika se često koristi u IT-u za pronalaženje najučinkovitije rute za putovanje podataka između više različitih čvorova. Aplikacije se sastoje od identificiranja mogućih optimizacija mrežnog ili hardverskog dijela. Prvi ga je opisao Irski matematičar W.R. Hamilton i britanski matematičar Thomas Kirkman u 1800-ima kroz stvaranje igre koja je bila rješiva pronalaženjem Hamiltonovog ciklusa, odnosno puta koji se ne preklapa, a prolazi kroz sve čvorove. Umjesto da se fokusira na pronalaženje najučinkovitije rute, problem trgovačkog putnika se često bavi pronalaženjem najjeftinijeg rješenja. U problemu trgovačkog putnika velika količina varijabli stvara izazov pri pronalaženju najkraće rute, najčešće je najkraća ruta ujedno i najjeftinija.

6.2 Google algoritam pretraživanja

Google-ov algoritam pretraživanja jedan je od najinovativnijih algoritama pretraživanja. Bez ikakve sumnje može se reći da je njihov algoritam uvelike utjecao na svijet. Algoritam je kombinacija više manjih algoritama koju uzimaju stotine raznih faktora u obzir kao što su spominjanje ključnih riječi, upotrebljivost web stranice i povratne veze. Nažalost, točni algoritmi koji se koriste nisu dostupni javnosti, Google-ovi programeri u prosijeku mijenjaju programski kod algoritma 6 puta dnevno. Kada kažemo algoritam pretraživanja mislimo na optimizaciju za tražilice (SEO[4]) - prva asocijacija su vjerojatno Googleovi faktori rangiranja. Drugim riječima, što Google gleda kada odlučuje koje će stranice rangirati i kojim redoslijedom? Ako pogledamo Googleovu stranicu "Kako funkcionira pretraživanje", ona izravno otkriva neke od Googleovih najistaknutijih faktora rangiranja:

1. Povratne veze: Linkovi iz provjerenog izvora koji vode na stranicu
2. Novitet: Google pretežito izbacuje članke objavljene unutar zadnjih 24 sata
3. Spominjanje ključnih riječi: Koliko puta se tražene riječi pojavljuju na web stranici koju tražimo
4. Korisničko iskustvo: Pristupačnost same web stranice i optimizacija
5. Topikalni autoritet: Web stranice koje služe pretraživanju širokom aspektu upita

[4] niz aktivnosti koje su usmjerene prema podizanju posjećenosti stranica s tražilica

7. Pregled navedenih algoritama

Svaki od navedenih algoritama ima svoju kompleksnost i vrijeme koje mu je potrebno da se izvrši. U ovom djelu rada ćemo na primjeru pokazati koliko je svakom od navedenih algoritama potrebno vremena, u sekundama, da se izvede. Svaki od algoritama će imati zadane iste parametre: nasumično odabranu vrijednost koju će tražiti i listu cijelih brojeva koja će se svakim ponavljanjem imati za 100 elemenata više. Svaki algoritam će biti izvršen 10 puta radi točnijeg prikazivanja podataka i lakšeg donošenja zaključka.

7.1 Linearno pretraživanje – primjer

Kako se broj elemenata povećava od 100 do 1000, maksimalni broj usporedbi potrebnih za linearno pretraživanje također raste linearno. Na primjer, s 300 elemenata, linearna pretraga bi zahtijevala najviše 300 usporedbi, s 400 elemenata zahtijevala bi najviše 400 usporedbi, i tako dalje. Vremenska složenost linearne pretrage ostaje O(n), gdje je n veličina niza.

Ukratko, kako se broj elemenata povećava, linearna pretraga postaje sporija jer zahtijeva veći broj usporedbi da bi se pronašao ciljni element. Vremenska složenost linearne pretrage ostaje linearna (O(n)), gdje je n veličina niza. Za veće nizove, drugi algoritmi pretraživanja kao što je binarno pretraživanje ili interpolacijsko pretraživanje mogu pružiti brže vrijeme pretraživanja iskorištavanjem određenih karakteristika niza, kao što je sortirani poredak ili distribucija.

Tablica 1 Linearno pretraživanje – primjer

Br. elemanata	Tražena vrijednost	Vrijeme izvđenja (s)	Index na kojem se nalazi tražena vrijednost
100	73	0.0010020732879638672	72
200	112	0.0010008811950683594	111
300	297	0.0010018348693847656	296
400	214	0.0009999275207519531	213
500	460	0.0009996891021728516	459
600	168	0.0010013580322265625	167
700	408	0.0010015964508056640	407
800	241	0.0010001659393310547	240
900	876	0.0010018348693847656	875
1000	654	0.0019996166229248047	653

Izvor: Izrada aurora

7.2 Binarno pretraživanje - primjer

Kada je veličina niza mala, kao u prvom slučaju sa 100 elemenata, binarno pretraživanje je i dalje učinkovito i može brzo pronaći ciljni element. Vremenska složenost algoritma binarnog pretraživanja

je O(log n), gdje je n veličina niza. U slučaju niza s 100 elemenata, algoritam će trebati najviše 7 iteracija da pronađe ciljni element, što je vrlo brzo.

S druge strane, u slučaju 1.000.000 elemenata, binarno pretraživanje je još uvijek vrlo učinkovito i može brzo pronaći ciljni element. U slučaju niza s 1.000.000 indeksa, algoritmu će trebati najviše 20 ponavljanja da pronađe ciljni element, što je još uvijek vrlo brzo.

Stoga je binarno pretraživanje učinkovito i za male i za velike nizove, a učinkovitije je od linearnog pretraživanja za sortirane nizove. Ima logaritamsku vremensku složenost, što ga čini bržim od linearnog pretraživanja kako se veličina niza povećava. Međutim, binarno pretraživanje zahtijeva da niz bude sortiran, što može dodati dodatni trošak u smislu vremena i memorije. Općenito, za velike nizove, binarno pretraživanje je bolji izbor od linearnog pretraživanja.

Tablica 2 Binarno pretraživanje – primjer

Br. elemenata	Tražena vrijednost	Vrijeme izvđenja (s)	Index na kojem se nalazi tražena vrijednost
100	86	0.0009984970092773438	85
200	196	0.0009994506835937500	195
300	222	0.0019950866699218750	221
400	147	0.0010001659393310547	146
500	188	0.0019986629486083984	187
600	414	0.0019998550415039062	413
700	44	0.0019991397857666016	43
800	108	0.0019981861114501953	109
900	594	0.0019979476928710938	593
1000	777	0.0009994506835937500	776

Izvor: Izrada autora

7.3 Jump search – primjer

Kada je veličina niza mala, kao u slučaju 100 elemenata, *jump search* možda neće biti znatno brže od linearnog ili binarnog pretraživanja. Broj blokova kreiranih možda neće biti dovoljan da pruži značajnu prednost nad ovim algoritmima, a linearno pretraživanje na svakom bloku može potrajati dosta vremena. U tom slučaju, binarno pretraživanje ili linearno pretraživanje može biti brže.

S druge strane, kada je veličina niza velika, kao u slučaju 1000 indeksa, *jump search* može biti vrlo brzo i učinkovito. Broj blokova stvorenih bit će puno veći, a linearnop retraživanje na svakom bloku trajat će kraće. Kao rezultat toga, *jump search* može biti znatno brže od linearnog pretraživanja za velike nizove.

Tablica 3. Jump pretraživanje – primjer

Br. elemenata	Tražena vrijednost	Vrijeme izvđenja (s)	Index na kojem se nalazi tražena vrijednost
100	45	0.0009999275207519531	44
200	167	0.0019989013671875000	166
300	220	0.000997781753540039	219
400	9	0.0009989738464355469	8
500	361	0.0009984970092773438	360
600	368	0.0010039806365966797	367
700	619	0.0009987354278564453	618
800	178	0.0010008811950683594	177
900	203	0.0009982585906982422	202
1000	904	0.0009968280792236328	903

Izvor: Izrada autora

Općenito, brzina *jump search* algoritma ovisit će o veličini niza koji se pretražuje. Za manje nizove, *jump search* možda neće pružiti značajnu prednost u odnosu na druge algoritme pretraživanja, dok za veće nizove, *jump search* može biti znatno brže od linearnog i binarnog pretraživanja.

7.4 Interpolation search algoritam

Kada je veličina niza mala, kao što je 100 indeksa, brzina interpolacijskog pretraživanja možda neće biti značajno brža od drugih algoritama pretraživanja. S druge strane, kada je veličina niza velika, kao što je 1.000.000 indeksa, interpolacijsko pretraživanje može biti prilično brzo. S velikim nizom, veća je vjerojatnost da ćete imati ravnomjerniju distribuciju vrijednosti, što omogućuje interpolacijsko pretraživanje za točniju procjenu položaja ciljanog elementa. Kao rezultat toga, interpolacijsko pretraživanje može pružiti brže vrijeme pretraživanja u usporedbi s drugim algoritmima, osobito kada je ciljni element bliži početku niza.

Tablica 4 Interpolacijsko pretraživanje – primjer

Br. elemenata	Tražena vrijednost	Vrijeme izvđenja (s)	Index na kojem se nalazi tražena vrijednost
100	85	0.0010008811950683594	84
200	101	0.0019993782043457030	100
300	245	0.0009987354278564453	244
400	103	0.0009951591491699219	102
500	345	0.0020022392272949220	344
600	427	0.0009996891021728516	436
700	527	0.0009996891021728516	526
800	320	0.0020005702972412110	319
900	57	0.0019948482513427734	56
1000	985	0.0010027885437011719	954

Izvor: Izrada autora

Sve u svemu, brzina algoritma interpolacijskog pretraživanja ovisi o veličini i distribuciji vrijednosti u nizu. Za manje nizove, interpolacijsko pretraživanje možda neće ponuditi značajnu prednost u odnosu na druge algoritme, dok za veće nizove s ravnomjernijom distribucijom može pružiti brže vrijeme pretraživanja.

7.5 Eksponencijalno pretraživanje

Eksponencijalno pretraživanje je učinkovit algoritam za pretraživanje u sortiranim nizovima. Kako se veličina polja povećava, povećava se i broj usporedbi koje zahtijeva eksponencijalno pretraživanje, ali logaritamskom brzinom. Dakle, vrijeme pretraživanja za eksponencijalno pretraživanje ostaje relativno brzo čak i s većim nizovima. U usporedbi s linearnim pretraživanjem, eksponencijalno pretraživanje omogućuje značajno poboljšanje vremena pretraživanja, posebno za veće nizove.

Tablica 5 Eksponencijalno pretraživanje – primjer

Br. elemenata	Tražena vrijednost	Vrijeme izvđenja (s)	Index na kojem se nalazi tražena vrijednost
100	33	0.0010004043579101562	32
200	191	0.0020005702972412110	190
300	78	0.0010008811950683594	77
400	362	0.0020015239715576170	361
500	225	0.0009953975677490234	224
600	571	0.0010020732879638672	570
700	507	0.0010013580322265620	506
800	739	0.0010044574737548828	738
900	442	0.0009999275207519531	441
1000	993	0.0010001659393310547	992

Izvor: Izrada autora

7.6 Fibonacci search algoritam

Kod liste sa 100 elemenata algoritam bi izračunavao Fibonaccijeve brojeve dok ne pronađe broj koji je veći ili jednak veličini niza. Zatim bi izvršio binarno pretraživanje koristeći Fibonaccijeve brojeve kao točke dijeljenja. Vremenska složenost Fibonaccijeve pretrage u ovom slučaju je O(log n), što je približno log(100) = 6. To znači da bi algoritam zahtijevao približno 6 usporedbi da pronađe ciljni element.

Sa 1000 elemenata, *Fibonacci search* ponovno bi izračunao Fibonaccijeve brojeve dok ne pronađe broj koji je veći ili jednak veličini niza. Zatim bi izvršio binarno pretraživanje koristeći Fibonaccijeve brojeve kao točke dijeljenja. Vremenska složenost Fibonaccijeve pretrage u ovom slučaju je približno log(1000) = 9. To znači da bi algoritam zahtijevao približno 9 usporedbi da pronađe ciljni element.

Tablica 6 Fibonaccijevo pretraživanje – primjer

Br. elemenata	Tražena vrijednost	Vrijeme izvđenja (s)	Index na kojem se nalazi tražena vrijednost
100	40	0.0009975433349609375	39
200	54	0.0009989738464355469	53
300	256	0.0010004043579101562	255
400	231	0.0009958744049072266	230
500	406	0.0009999275207519531	405
600	598	0.001001119613647461	597
700	629	0.0010001659393310547	628
800	726	0.0009989738464355469	725
900	871	0.0009956359863281250	870
1000	977	0.0019981861114501953	976

Izvor: Izrada autora

Ukratko, Fibonaccijevo pretraživanje učinkovit je algoritam pretraživanja sortiranih nizova. Kako se veličina niza povećava, povećava se i broj usporedbi potrebnih za Fibonaccijevo pretraživanje, ali logaritamskom brzinom. Vrijeme pretraživanja za Fibonaccijevo pretraživanje ostaje relativno brzo čak i s većim nizovima. Međutim, u usporedbi s drugim algoritmima pretraživanja poput binarnog pretraživanja ili interpolacijskog pretraživanja, Fibonaccijevo pretraživanje nije najučinkovitiji izbor.

8. Zaključak

Zaključno, algoritmi pretraživanja služe kao nezamjenjivi alati u području Python programiranja, olakšavajući učinkovito i djelotvorno pronalaženje informacija iz različitih struktura podataka. Python nudi raznolik raspon algoritama pretraživanja, svaki sa svojim skupom prednosti i razmatranja. Jednostavnost linearnog pretraživanja čini ga jednostavnom opcijom za male pretrage, iako s linearnom vremenskom složenošću. S druge strane, binarno pretraživanje, sa svojom logaritamskom vremenskom složenošću, pokazalo se vrlo učinkovitim kada se radi sa sortiranim podacima. Pythonova svestranost blista kroz algoritme kao što su interpolacijsko pretraživanje i *jump serach*, koji se prilagođavaju različitim scenarijima. Interpolacijsko pretraživanje koristi distribuciju elemenata i tehnike interpolacije kako bi se postigla ravnoteža između vremenske složenosti i izvedbe. U međuvremenu, *jjump search* jako dobro radi dijeljenjem podataka u blokove i korištenjem fiksne veličine koraka za optimizaciju procesa pretraživanja. Razumijevanjem prednosti i ograničenja ovih algoritama, Python programeri stječu znanje za donošenje informiranih odluka na temelju jedinstvenih karakteristika svojih podataka i specifičnih zahtjeva njihovih aplikacija. Kroz vještu upotrebu ovih algoritama pretraživanja, programeri Pythona mogu maksimalno povećati učinkovitost operacija pretraživanja, poboljšavajući ukupnu izvedbu svojih programa u stvarnim aplikacijama koje obuhvaćaju baze podataka, web tražilice, aplikacije za sortiranje i pretraživanje, računalno umrežavanje, pa čak i domene umjetne inteligencije .

Stoga pravilno razumijevanje algoritama pretraživanja u Pythonu omogućuje programerima da se brzo i učinkovito kreću kroz ogromne skupove nesortiranih podataka, otvarajući vrata bržim, učinkovitijim i robusnijim rješenjima.

9. Izjava

Izjava o autorstvu završnog rada i akademskoj čestitosti

Ime i prezime studenta: Marko Kanceljak

Matični broj studenta: 0016133603

Naslov rada: PREGLED I USPOREDBA ALGORITAMA ZA PRETRAŽIVANJE

Pod punom odgovornošću potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature, ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada.

Potvrđujem da je elektronička verzija rada identična onoj tiskanoj te da je to verzija rada koju je odobrio mentor.

Datum Potpis studenta

_____ _____

10. Literatura

1. A. Ivančić, *Usporedba algoritma za sortiranje i pretraživanje,* URL: Usporedba algoritma za sortiranje i pretraživanje, pristupano 17.03.2023
2. A. Levitin, 2012, Introduction to The Design and Analysis of Algorithms. USA: Pearson
3. A. Sweigart, 2015, Automate the Boring Stuff with Python
4. C. Giridhar, 2010, Python 3 Object-Oriented Programming
5. Codesera, Kela Casy, *Let Us Understand Searching Algorithms,* URL: Let Us Understand Searching Algorithms - CODERSERA, pristupano 17.03.2023
6. Crazy domains.com, Patrick Compton, *The History of Search Engines and Their Algorithms: How Do They Impact SEO,* URL: The History of Search Engines and Their Algorithms: How Do They Impact SEO - Crazy Domains Learn, pristupano 20.12.2022
7. Geekg for geeks, *Binary search,* URL: Binary Search - GeeksforGeeks, pristupano 17.03.2023
8. Geekg for geeks, *Exponencial search,* URL: Exponencial Search - GeeksforGeeks, pristupano 20.12.2022
9. Geekg for geeks, Fibonacci *search,* URL: Fibonacci Search - GeeksforGeeks, pristupano 20.12.2022
10. Geekg for geeks, *Jump search,* URL: Jump Search - GeeksforGeeks, pristupano 20.12.2022
11. Geekg for geeks, *Linear search,* URL: Linear Search - GeeksforGeeks, pristupano 17.03.2023
12. Geekg for geeks, *Seraching algorithms,* URL: Searching Algorithms - GeeksforGeeks, pristupano 17.03.2023
13. Google Search - Discover How Google Search Works pristupano 20.12.2022
14. How the Google Search Algorithm Works (ahrefs.com) pristupano 20.12.2022
15. J. Mosher, *Search Algorithms,* URL: Search Algorithms, pristupano 20.12.2022
16. M. Varga, 2018, Programiranje u programskim jezicima Phyton i Visual Basic,
17. M. Vujošević Janičić i J. Graovac, *Složenost algoritma*, URL: Složenost, pristupano 03.09.2022
18. Nova akropola, *Zlatni rez*, URL: Nova Akropola – Zlatni rez (nova-akropola.com), pristupano 17.03.2023
19. R. Manger, 2014, Strukture podataka i algoritmi. 1st ur. Zagreb: ELEMENT d.o.o.
20. R. Sedgewick i K. Wayne, 2011, *Algorithms, fourth edition,* URL: Algortihms, pristupano 05.04.2023
21. T. H. Cormen i suradnici, 2009, *Introduction to Algoritghms, third edition,* URL: Introduction to Algoritghms, pristupano 06.04.2023
22. W. McKinney, 2012 , Python for Data Analysis pristupano 06.04.2023
23. Whatls.com, traveling salesman problem, URL: TSP, pristupano 06.04.2023

11. Popis slika

Slika 1 Linearno pretraživanje u Phytonu 11
Slika 2 Linearno pretraživanje 12
Slika 3Prvi korak binary search 13
Slika 4 Binary pokazivači 13
Slika 5 Binary pokazivači 2.korak 14
Slika 6 Binary kraj 14
Slika 7 Binarno pretraživanje u Phytonu 15
Slika 8 Jump pretraživanje u phytonu 17
Slika 9 Algoritam pretraživanja interpolacijom pisan u Phytonu 19
Slika 10 Algoritam eksponencijalnog pretraživanja pisan u Phytonu 21
Slika 11 Fibbonachi - 1 22
Slika 12 Fibbonachi - 2 23
Slika 13 Algoritam Fibonaccijevog pretraživanja u Phytonu 25
Slika 14 Algoritam hash pretraživanja u Phytonu 26

12. Popis tablica

Tablica 1 Linearno pretraživanje – primjer 31
Tablica 2 Binarno pretraživanje – primjer 32
Tablica 3 Jump pretraživanje – primjer 33
Tablica 4 Interpolacijsko pretraživanje – primjer 34
Tablica 5 Eksponencijalno pretraživanje – primjer 34
Tablica 6 Fibonaccijevo pretraživanje – primjer 35

Marko Kanceljak

Datum rođenja: 17/06/1998 | **Državljanstvo:** hrvatsko | **Spol:** Muško | **Telefonski broj:** (+385) 917963851 (Mobilni telefon) | **E-adresa:** marko.kanceljak98@gmail.com |

Adresa: Pandaki 34, 10257, Kupinečki kraljevec, Hrvatska (Kućna)

OBRAZOVANJE I OSPOSOBLJAVANJE

01/09/2013 – 01/06/2017 Zagreb, Hrvatska
TEHNIČAR ZA RAČUNARSTVO Elektrotehnička škola u Zagrebu

01/09/2019 – TRENUTAČNO Zagreb, Hrvatska
PRVOSTUPNIK INFORMACIJSKIH TEHNOLOGIJA Veleučilište s pravom javnosti BALTAZAR ZAPREŠIĆ

Internetske stranice https://www.bak.hr/

JEZIČNE VJEŠTINE

Materinski jezik/jezici: **HRVATSKI**

Drugi jezici:

	RAZUMIJEVANJE		GOVOR		PISANJE
	Slušanje	Čitanje	Govorna produkcija	Govorna interakcija	
ENGLESKI	C1	C1	C1	C1	C1

Razine: A1 i A2: temeljni korisnik; B1 i B2: samostalni korisnik; C1 i C2: iskusni korisnik

DIGITALNE VJEŠTINE

Rad na raunalu | Timski rad | Windows | Internet | Informacije i komunikacija (pretraivanje interneta) | Komunikacijski programi (Skype Zoom TeamViewer) | MS Office (Word Excel PowerPoint)

RADNO ISKUSTVO

01/04/2020 – TRENUTAČNO Zagreb, Hrvatska
SALES ADMINISTRATOR AUTO BENUSSI D.O.O.

- Zaprimanje i fakturiranje novih i rabljenih vozila
- Pribavljanje potrebne dokumentacija za registraciju vozila
- Pribavljenje i priprema dokumentacije za registraciju vozila iz inozemstva
- Održavanje lagera
- Oglašavanje vozila

DODATNE INFORMACIJE

VOZAČKA DOZVOLA

Vozačka dozvola: AM

Vozačka dozvola: A1

Vozačka dozvola: A2

Vozačka dozvola: A

Vozačka dozvola: B