UNIVERZITET U ZENICI

POLITEHNIČKI FAKULTET

SOFTVERSKO INŽENJERSTVO

Seminarski rad

Ricochet Robots

Tim:

Amer Bužo, 215.

Orhan Pojskić 218.

Ajla Brdarević, 219.

Amar Kamberović, 220.

Mentori:

prof. doc. dr Nermin Goran

V. Asistent Sanid Muhić

S.P. Lejla Mašić

Predmet: Algoritmi i strukture podataka

Zenica, 2022

# UVOD

Tema našeg seminarskog rada iz predmeta Algoritmi i strukture podataka je igra Ricochet Robots. U nastavku seminarskog rada prije svega ćemo objasniti pravila kojima se igra Ricochet Robots i koji je cilj.

Ono što je zanimljivo kod ove igre je što je napravljena tako da testira, kako ljude tako i računare, način rješavanja problema, posebno kad su u pitanju algoritmi. Igra na prvu zvuči jako jednostavno - pomjerati figure tj. robote sa jednog polja na drugo. No, međutim, postoji cijela procedura iza toga kako doći do cilja. Mnogima je baš Ricochet Robots bila jako zanimljiva u akademskim istraživanjima upravo zbog svoje kompleksnosti. Postoji kao online igrica kako je prvobitno izdata 1999. godine i kao real life igrica. Danas ima brojne “nasljednike” i nove verzije.

Ricochet Robots testira vještinu rješavanja problema i strpljivost kod igrača, jer tek tokom igranja shvatimo koliko opcija za pomeranje robota postoji, i kako nijedna od tih opcija nije optimalna, jer uvijek postoji način da se brže i lakše dođe do cilja. Također, u nastavku ćemo objasniti po kojem principu i algoritmu radi, te na koji način možemo odlučiti kako ispuniti cilj igrice. http://www.robotreboot.com/classic



Slika 1. Ricochet Robots

# 

# SADRŽAJ

[UVOD](#_21p782de92f3) 2

[SADRŽAJ](#_cpnu07kvr4nj) 3

[RICOCHET ROBOTS](#_ddediebal57g) 4

[KAKO SE IGRA](#_s2e9p5qfpl2r) 4

[PROBLEM](#_zcdzbwbjtsw8) 5

[POLINOMSKI VREMENSKI ALGORITAM](#_upaurevvk78i) 5

[RJEŠAVANJE PROBLEMA](#_6oa1s5kjsdje) 6

[KRAJ](#_jcfylu7k45o1) 12

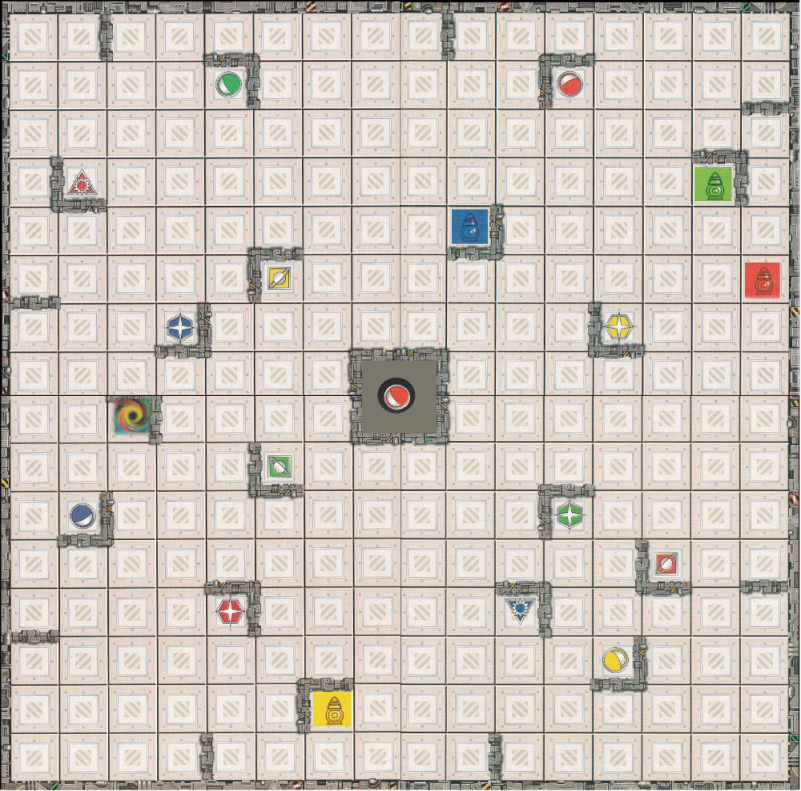
[LITERATURA](#_5kybl252tm8f) 13

# RICOCHET ROBOTS

Ricochet Robots je slagalica koju mogu igrati 2 ili više ljudi. Dizajnirao je Alex Randolph, te je objavljena u Njemačkoj 1999. godine kao Rasende Roboter. Cilj slagalice je da se roboti sa kojim se igra, premjeste na odabrana mjesta u što je moguće manje poteza, s tim što postoje stroga ograničenja kretanja robota

### KAKO SE IGRA

Slagalica se sastoji od ploče dimenzija 16x16 sa 4 ili 5 robota različitih boja. Na ploči se nalaze mete označene bojama. Roboti su postavljeni na mjesta koja odaberu igrači. Mogu se kretati poput topova u šahu (polja ravno ispred sebe, iza sebe, te lijevo i desno od sebe) sve dok ne naiđu na prepreku - zid ili drugog robota. Cilj slagalice je kako smo već rekli, dovesti “aktivnog” robota jedne boje do ciljanog kvadrata te boje (tj. do nasumično odabrane mete iste boje kao i robot) u što manje poteza. Nakon što se izvuče kartica ili žeton koji određuje šta je meta i koje boje, te koje boje robot treba da dođe do te mete, igra započinje. Dakle, kada se robot jednom počne kretati neće stati sve dok ne naiđe na prepreku ili drugog robota. To znači da igrači traže slijed poteza za robote koji će im omogućiti da pomaknu traženog robota do mete u najmanje poteza. Neaktivni roboti se mogu koristiti kao prepreke, tako da ako imamo na primjer robote plave, crvene, žute i zelene boje, a izvučemo žeton sa metom plave boje te treba odvesti plavog robota do te boje, preostala 3 robota treba postaviti da su nam prepreke/barijere kako bi mogli u što manje koraka doći do plave mete.



Slika 2. Ploča za Ricochet Robots

### PROBLEM

Problem: Kako pomjeriti robota do ciljane mete u što manjem broju poteza?

Struktura ove slagalice ima dosta složenu tehniku rješavanja problema, stoga je jako zanimljiva sa gledišta rješavanja preko algoritama te za testiranje vještine rješavanja složenih problema kako ljudi tako i računara.

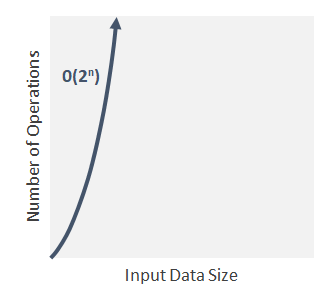
Ono što smo da sada mogli primjetiti kroz proučavanje pravila igre je to da igra u početku ima eksponencijalno povećanje složenosti[[1]](#footnote-2), što je rješivo polinomskim vremenskim algoritmom[[2]](#footnote-3). Broj mogućih pozicija je uvijek ograničen relativno malim brojem robota (četiri ili pet robota) te takav algoritam ima mogućnost da pretraži sve te moguće pozicije.

### POLINOMSKI VREMENSKI ALGORITAM

Algoritam polinomskog vremena je algoritam čije je vrijeme izvršavanja ili zadano polinomom veličine ulaza, ili može njime da bude ograničeno. Problemi koji su rješivi ovim algoritmom se nazivaju “*tractable* problems” (rješivi problemi). Većina algoritama na nizovima koristi n - veličinu niza, kao ulaznu veličinu. Na primjer za pronalaženje najvećeg elementa u nizu se koristi algoritam O(n) ili linearno vrijeme. Dok algoritmi za sortiranje obično zahtijeva ili O (n logn) ili O(2n). Dakle, kod ovog algoritma broj koraka za rješavanje problema je ograničen polinomskom funkcijom od n, gdje n odgovara dužini ulaznog podatka.

U našem slučaju, broj mogućih pozicija je ulazni podatak n, što znači da je broj koraka algoritma ograničen polinomskom funkcijom od mogućih pozicija. U početku imamo eksponencijalni rast složenosti jer se stopa rasta udvostručuje svakim novim dodatnim ulazom (n), to jeste svakom novom opcijom koju otkrijemo za pomjeranje robota udvostručuje se broj mogućih poteza. Svaki put kada nam se broj pozicija - ulaznih podataka poveća za jedan, to nam automatski udvostručuje broj izvršenih operacija.

Dakle, da bi riješili eksponencijalno povećanje složenosti koji se dugo rješava, koristimo polinomski vremenski algoritam jer može da riješi puno veći opseg i broj problema za kraće vrijeme.



Slika 3. Eksponencijalno povećanje složenosti

### RJEŠAVANJE PROBLEMA

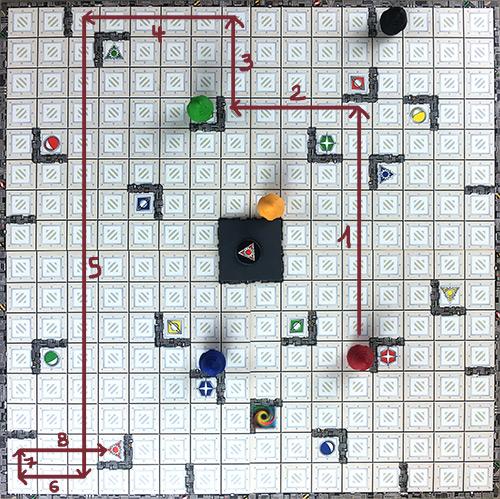
Prije stvaranja samog algoritma moramo sagledati par stvari i problema koji utječu na algoritam. Gledajući u ploču, prvo što vidimo su polja. Već tu nailazimo na prvi faktor koji utječe na najoptimalniji način dolaska do cilja.

1. Veličina ploče je kako već znamo 16x16. To znači da imam 256 polja, s tim što centralno područje zauzima 4 polja, pa prema tome imamo 252 polja i 5 aktivnih robota. Dakle broj mogućih pozicija robota bi bio:

*252\*251\*250\*249\*248 = 976 484 376 000*

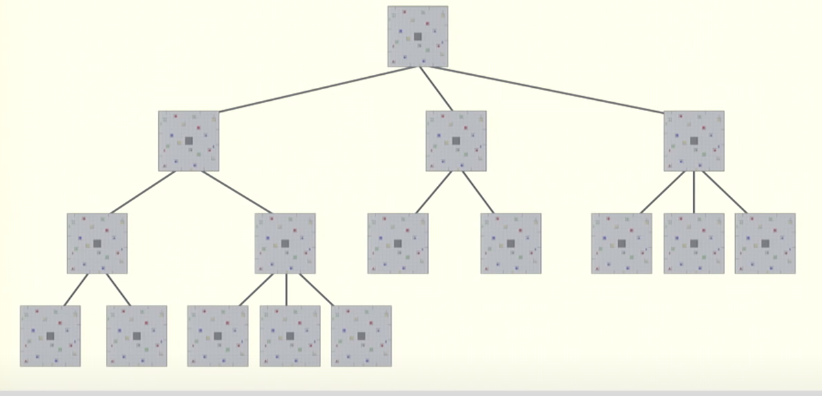
Upravo iz ovog izračuna vidimo šta je kompleksnost igrice. Postoji veliki broj mogućih pozicija svakog robota.

1. Drugi problem nam predstavlja faktor grananja iz svakog stanja ploče, to jeste pozicije. Postoji od 9 do 20 mogućih pomicanja iz svakog stanja. Zbog velikog broja stanja (pozicija) to je jako veliki broj mogućnosti. Ovaj problem moramo razložiti kako bi razumjeli šta se to zapravo mijenja u toku igre i koliko često.
2. Imamo ploču koja je statički dio. Statički je dio jer je uvijek ista i na njoj se ništa ne mijenja tokom igre.
3. Imamo prepreke i mete. One su relativno statičke u smislu da se mijenjaju iz igre u igru, ali ne i tokom jedne igre. Tako nam ni ovaj faktor ne predstavlja veliki problem jer je posmatrajući jednu igru statički.
4. Cilj se mijenja kroz igru, ali je također relativno statičan element kada posmatramo jedan ciklus igre. U svakom slučaju je manje statičan od prepreka i meta, pa nam predstavlja faktor koji utječe na algoritam rješavanja.
5. Pozicija robota se mijenja tokom svakog poteza. To je dio koji stvara najveći problem jer ima mnogo opcija.



Slika 4. Graf mogućih poteza

Sada kada smo shvatili koje su komponente igre i šta je to što predstavlja najveći problem, možemo zaključiti da su nam stanja na ploči zapravo kombinacije ciljanog polja i pozicije robota. Također, treba da predstavimo kretanja robota. Sve to možemo predstaviti stablom. Stanja na ploči su čvorovi drveta i svaka ivica drveta je kretanje robota u određenom pravcu. Ono što također trebamo uzeti u obzir je da svaki čvor ima roditelja. Glavni čvor na početku nema roditelja, a svaki drugi čvor ima.

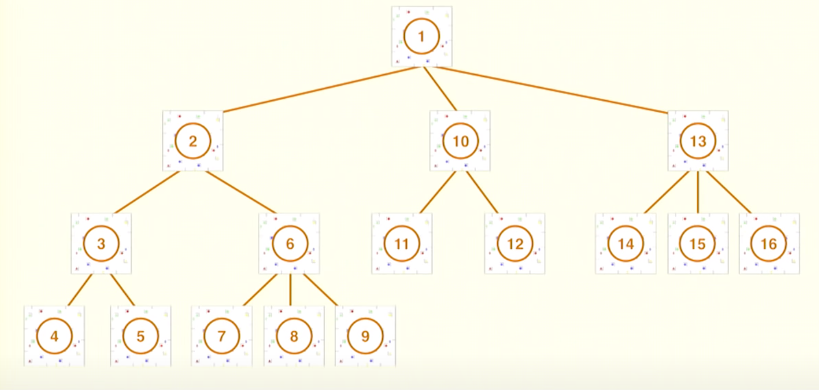


Slika 5. Stablo

Sada kada smo stvorili stablo, ono što se obično radi, a što ćemo i mi iskoristiti je pretraživanje, koje je obično prvi korak za ovakve algoritme. Počet ćemo od koraka koji na prvu ne izgleda kao optimalan, ali ga moramo uzeti u obzir kako bi shvatili koji su dodatni faktori koji utječu na razvijanje algoritma.

1. The Depth-First Search (“Dubinsko pretraživanje”)

Prvo da se podsjetimo dubinskog pretraživanja. Pretražujemo niz jednu cijelu granu stabla počevši od glavnog čvora, sve dok ne dođemo do kraja. To bi izgledalo kao na slici 6.

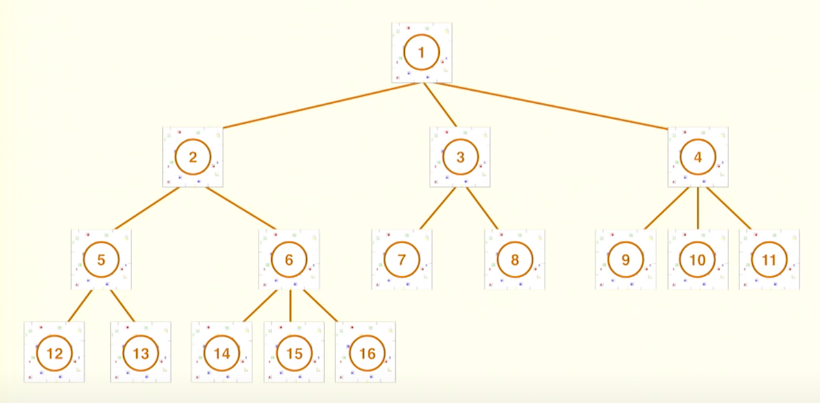


Slika 6. Dubinsko pretraživanje - The Depth-First Search

Ovdje nailazimo na problem. To je mogućnost da robot “zaglavi” u beskonačnom krugu iz kojeg ne može nikada da izađe. Sljedeći logičan korak bi bio da izostavimo stanja u kojima bi došlo do kretanja robota u beskonačnom krugu, ali u većini slučajeva dolazak na svako polje je moguć sa dva puta. Zbog toga stablo koje smo pokazali na slikama 5. i 6. zapravo nije stablo, nego graf gdje čvorovi imaju više od jednog roditelja. Ovdje vidimo još jedan problem koji je povezan sa pravilima Ricochet Robots-a. Ovaj tip algoritma nije najbolje rješenje jer postoje neke solucije koje su pravilima zabranjeni, te ih moramo izuzeti. Također, imamo ogroman broj opcija (976 484 376 000) koje moramo pretražiti, što nije optimalno vremenski niti memorijski. Sada kada smo shvatili putem stabla na koje prepreke nailazimo, razradit ćemo problem na malo drugačiji način.

1. Breadth-First Search

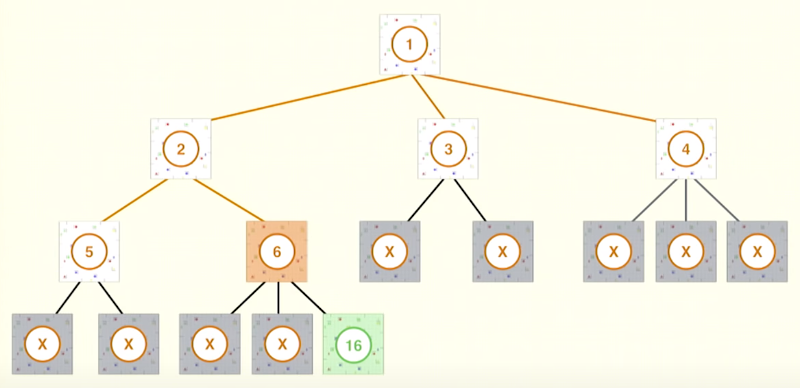
U ovom pretraživanju, umjesto pretraživanja niz jednu cijelu granu stabla, pretražujemo po razinama stabla kao na slici 7.



Slika 7. Breadth-First Search

Ovakav tip algoritma definitivno osigurava pronalaženje najkraće putanje. Ovo radi na principu da prvu putanju stavljamo u red čekanja, ako ne ispunjava cilj ide na kraj reda, a ako je opcija koja ispunjava cilj vraća podatak. Breadth-First Search nam već skraćuje vrijeme pretraživanje jer ne mora pretraživati ista stanja više puta kao u dubinskom pretraživanju. No, to je i dalje vremenski neprihvatljiv način. Zbog toga moramo nešto optimizirati. Prije svega treba razmisliti o tome kako da algoritam radi manje tj. da reducira prostor za pretragu tako što se stablo “reže”. Zatim trebamo razmisliti kako da algoritam radi stvari brže, odnosno da radi manje posla po svakom stanju. Možemo iskoristiti i heuristički način pristupa. Tu bi prvo trebali posmatrati aktivnog robota jer nam je to možda najbrži način dolaska do ciljanog polja. Uzeći u obzir brojne faktore koje smo već naveli, ovo bi također bio ne-efektivan način pristupa.

Najveći problem na koji nailazimo je dugo vrijeme pretraživanja, što se dešava zbog toga što imamo previše objekata koje posmatramo. Vratit ćemo se na optimizaciju da algoritam radi manje posla. Pod tim podrazumijevamo da nađemo način da prije svega smanjimo broj objekata koje uključujemo u pretraživanje.



Slika 8. Smanjivanje broja objekata za pretraživanje

Na slici 8. možemo vidjeti šta smatramo smanjivanjem objekata. Za razliku od dosadašnjih grafova, ovdje smo smanjili broj koraka dolaska do čvora 16. Isključili smo sve grane i čvorove koji nisu povezani sa čvorom 16. To znatno smanjuje vrijeme potrebno za izvršavanje, pogotovo kad posmatramo mnogo brojnija stabla od navedenog.

Sada kada smo naveli veliki broj problema i samo neka od rješenja, koja nisu optimalna, šta je zapravo najbolje rješenje. S obzirom na količinu informacija i kompleksnost svega s čime smo se upoznali, teško je reći šta je najbolje od najboljih rješenja. Naišli smo i na brojne mogućnosti rješavanja na dosta kompleksnijem nivou, nama za sad dosta teško za razumjeti i implementirati. Zbog toga smo u našem kodu predstavili algoritme koje smo spomenuli do sada The Depth-First Search i Breadth-First Search.

Na slici 9. je naša implementacija dubinskog pretraživanja po uzoru na sliku 6. Prvi dio koda *graph* sadrži čvorove grafa. Prvi čvor je parent i ima 3 child čvora (2, 10 i 13). I tako za svaki čvor roditelj - dijete. U dijelu *visited = set()* je lista služi kako bi set koji smo postavili pratio koji čvorovi su posjećeni a koji ne. U funkciji *dfs* parametri su *visited* - služi da prati kojo čvorovi su posjećeni, *graph* - graf i *node* - čvorovi koji se nalaze u grafu. U uvjetu *if* pronalazi čvorove koji su posjećeni pomoću *for petlje*.

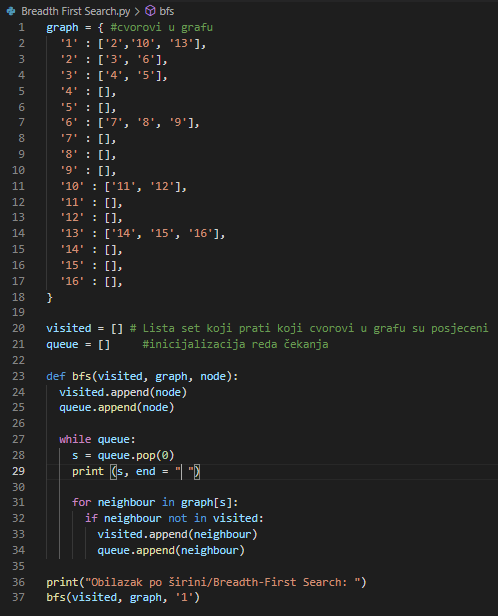


Slika 9. Depth First Search - implementacija



Slika 10. Ispis u terminalu za Depth First Search

Na slici 11. je implementacija pretraživanja po širini. Opet imamo *graph* gdje se nalaze svi čvorovi grafa, te *visited*  kao u prethodnom primjeru. Inicijalizaciju reda čekanja čvorova smo predstavili kao *queue = []*. U funkciji *bfs* sa parametrima *visited, graph* i *node* dodajemo .*append* za *visited* i *queue* da bi se provjerio i uvećao početni čvor. To se sve dešava dok god ima čvorova u redu čekanja koji nisu posjećeni.



Slika 11. Implementacija Breadth First Search



Slika 12. Ispis u terminalu za Breadth First Search

# 

# KRAJ

Kao što smo već i naveli, postoje mnogobrojna rješenja i implementacije koje služe za rješavanje Ricochet Robots igrice. Većina njih koja su vezana konkretno za igricu su dosta kompleksna kao i sama igrica. Jednostavniji algoritmi nisu u stanju da ispune cilj - da obrade toliko podataka za što kraće vrijeme. Jedno od zanimljivih rješenja na koje smo naišli - [https://github.com/fogleman/Ricoche](https://github.com/fogleman/Ricochet)t. U ovom radu smo se bazirali na neke jednostavnije ideje, i nama prihvatljivije s obzirom na naše iskustvo sa algoritmima.

Ricochet Robots je jako zanimljiva igrica, ili tačnije slagalica, koja iskušava limite algoritama. Sa mnogobrojnim preprekama koje smo naveli, i još mnogo njih koje smo izostavili kako bi shvatili poentu, vidimo zašto je čest primjer za istraživanja. Od brojnih već napravljenih algoritama za rješavanje, do brojnih koji se tek razvijaju, mogih izvora gdje kodovi rade i ne rade, istraživati o ovoj igrici je bilo izazovno. I nakon bezbroj puta igranja online verzije Ricochet Robots-a, nismo mogli pobijediti računar i algoritam. Tada smo i shvatili da je ovaj zadatak puno kompleksniji nego što smo na početku mislili, jer nekako se svaki put može doći bližim i bržim putem.

# 

# LITERATURA

1. <https://speakerdeck.com/fogleman/ricochet-robots-solver-algorithms?slide=2> (27/12/2021)
2. <https://boardgamegeek.com/boardgame/51/ricochet-robots> (27/12/2021)
3. <https://www.youtube.com/watch?v=OQOMmftaWAQ> (27/12/2021)
4. <https://towardsdatascience.com/essential-programming-time-complexity-a95bb2608cac> (30/12/2021)
5. <https://www.seas.upenn.edu/~cit596/notes/dave/p-and-np2.html> (9/1/2022)
6. <https://www.britannica.com/science/polynomial-time-algorithm> (9/1/2022)
7. <https://www.youtube.com/watch?v=fvuK0Us4xC4> (14/1/2021)

1. O(2n) - Algoritmi s ovom vremenskom složenošću obično se koriste u situacijama kao što je ova, kada ne znamo šta je to najbolje rješenje i moramo isprobati svaku moguću kombinaciju ili permuntaciju podatka [↑](#footnote-ref-2)
2. Algoritam polinomskog vremena je algoritam čije je vrijeme izvršavanja ili zadano polinomom veličine ulaza, ili može njime da bude ograničeno. [↑](#footnote-ref-3)