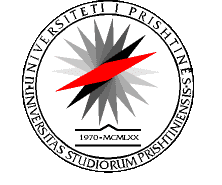
**UNIVERSITETI I PRISHTINËS “HASAN PRISHTINA”**

**FAKULTETI I INXHINIERISË ELEKTRIKE DHE KOMPJUTERIKE**

**DEPARTAMENTI: INXHINIERIA KOMPJUTERIKE – BACHELOR**



**Lënda**: Dizajni dhe Analiza e Algortimeve

**Tema**: Activity Selection Problem

**Studentët**:                                                             **Mentorë**:

Egzona Vllasaliu       Prof.Ass.Dr. Avni Rexhepi

Butrint Beqiri Msc. Dardan Shabani

Prishtinë, Maj 2020

Përmbajtja

[1. Hyrje 3](#_Toc39597964)

[1.1 Programimi dinamik 3](#_Toc39597965)

[1.2 Algoritmi lakmitar 3](#_Toc39597966)

[1.3 Dallimet mes programimit dinamik dhe algoritmit lakmitar 4](#_Toc39597967)

[2. Activity Selection Problem 5](#_Toc39597968)

[2.1 Qasja me programimin dinamik 6](#_Toc39597969)

[2.2 Qasja me algoritmin lakmitar 7](#_Toc39597970)

[2.3 Algoritmi lakmitar rekurziv 8](#_Toc39597971)

[2.4 Algoritmi lakmitar iterativ 9](#_Toc39597972)

[2.5 Kompleksiteti kohor 10](#_Toc39597973)

[2.6 Aplikimi i Activity Selection Problem 11](#_Toc39597974)

[3. Bllok-diagrami i algoritmit 12](#_Toc39597975)

[4. Implementimi 13](#_Toc39597976)

[Referencat 14](#_Toc39597977)

# 1. Hyrje

Në matematikë, shkencë kompjuterike dhe ekonomi, **problem i optimizimit** quhet problemi i përcaktimit të zgjidhjes më të mirë ose efektive nga të gjithë zgjidhjet e mundshme. Formalisht, problemi i optimizimit përbëhet nga tre pjesë:

* Një bashkësi joboshe e *instancave* **I**,
* Një *funksion objektiv* **v(x,y)**, ku **y** është një zgjidhje e mundshme për **x ∈ I**,
* Një *synim* që ose minimizon ose maksimizon funksionin objektiv.

Çdo instancë **x ∈ I** ka një dyshe të renditur (**Sx, OPT(x)**) ku **Sx** është bashkësia e zgjidhjeve të mundshme, ndërsa **OPT(x)** është **vlera optimale objektive** ose *vlera më e mirë ose e favorshme që synohet ose kërkohet*. Nëse një zgjidhje në **Sx** ka vlerën optimale objektive **OPT(x)**, ajo quhet **zgjidhje optimale**.

Algoritmet e problemeve të optimizimit zakonisht kalojnë nëpër një sekuencë hapash. Në çdo hap, për të arritur tek zgjidhja optimale, kanë disa zgjedhje që duhet të bëjnë. Përcaktimi i zgjedhjeve më të mira mund të bëhet përmes **programimit dinamik** ose **algortimeve lakmitare**.

## 1.1 Programimi dinamik

Programimi dinamik, sikur metoda përça-dhe-sundo, zgjidh problemin duke kombinuar zgjidhjet e nënproblemeve.

Metoda përça-dhe-sundo e copëzon problemin në nënprobleme disjunkte, zgjidh nënproblemet rekurzivisht dhe pastaj kombinion zgjidhjet në zgjidhjen përfundimtare të problemit origjinal.

Programimi dinamik aplikohet në rastet kur nënproblemet kanë pjesë të përbashkëta, d.m.th. kur nënproblemet ndajnë nënnënprobleme. Në këto raste, algoritmet përça-dhe-sundo bëjnë punë më shumë se që duhet, duke zgjidhur vazhdimisht nënnënproblemet e përbashkëta. Algoritmet e programimit dinamik zgjidhin çdo nënnënproblem vetëm njëherë dhe ruajnë përgjigjen në një tabelë, duke shmangur kështu përsëritjen e procesit për nënnënproblemet e njëjta.

Programimi dinamik aplikohet tek problemet e optimizmit pasi që këto probleme mund të kenë më shumë se një zgjidhje. Çdo zgjidhje ka një vlerë dhe ne duam të zgjedhim zgjidhjen më optimale të problemit. Megjithatë, për shumë probleme të optimizimit, përdorimi i programimit dinamik për të bërë zgjedhjen më të mirë, kushton shumë sa i përket kohës së ekzekutimit. Ekzistojnë algoritme më të thjeshta dhe efikase që mund të përdoren.

## 1.2 Algoritmi lakmitar

Algoritmi lakmitar (ang. greedy algorithm) gjithmonë bën zgjedhjen që duket më e mira në moment të caktuar. D.m.th., bën një zgjedhje lokale të vlerës optimale objektive me shpresën se ajo zgjedhje do të drejtojë tek zgjidhja optimale globale.

Algoritmet lakmitare jo gjithmonë japin zgjidhje optimale, por për shumë probleme funksionojnë.

Supozojmë se kemi një funksion objektiv që ka nevojë të optimizohet në një pikë të caktuar. Algoritmi lakmitar bën zgjedhje lakmitare në çdo hap për të sigurar që funksioni objektiv është optimizuar. Ky algoritëm ka vetëm një shans të bëj zgjedhjen drejt zgjidhjes optimale, asnjëherë nuk kthehet prapa që të tërheq vendimin e marur. Kjo do të thotë që algoritmi bën zgjedhjen më të mirë në hapin specifik që gjendet, pa marur parasysh atë që pason. Pra, zgjedhja bëhet asisoj që sjell përfitim të atypëratyshëm, pa u konsideruar pamja e plotë, zgjidhja finale.

## 1.3 Dallimet mes programimit dinamik dhe algoritmit lakmitar

1. Algoritimi lakmitar përcakton zgjidhjen optimale lokale në çdo hap me shpresën që do të drejtohet tek zgjidhja optimale globale. Programimi dinamik e ndan problemin në varg të nënproblemve me pjesë të përbashkëta dhe pastaj i zgjidh ato.
2. Algoritimi lakmitar kurrë nuk rishikon zgjedhjet e bëra, ndërsa programimi dinamik mund të konsiderojë gjendje të mëparshme.
3. Algortimi lakmitar është më pak efikas se programimi dinamik.
4. Algoritmi lakmitar bën zgjidhje lokale të nënproblemeve, kurse programimi dinamik i zgjidh të gjitha nënproblemet dhe pastaj zgjedh ato që drejtojnë tek zgjidhja optimale.
5. Algoritmi lakmitar punon në bazë të vetisë së zgjedhjes, kurse programimi dinamik punon me parimin e optimitetit.
6. Algoritimi lakmitar ndjek strategjinë lartë-poshtë kurse programimi dinamik ndjek strategjinë poshtë-lartë.

Programimi dinamik zgjidh nënproblemet **poshtë-lartë**. Kjo nënkupton që problemi nuk mund të zgjidhet deri sa të gjejmë të gjitha zgjidhjet e nënproblemeve. Zgjidhja optimale arrihet kur shfaqet i gjithë problemi.

Algoritmi lakmitar zgjidh nënproblemet nga **lartë-poshtë**. Pra, fillimisht bëhet zgjedhja lakmitare, e pastaj problemi reduktohet në problem më të vogël. Zgjidhja optimale mund të arrihet kur zhduket i gjithë problemi.

Programimi dinamik duhet të provojë çdo mundësi para se të arrijë tek zgjidhja. Kjo rrit kompleksitetin kohor.

Janë disa probleme që algoritmet lakmitare nuk mund t’i zgjidhin kurse programimi dinamik po. Prandaj, fillimisht provojmë algoritmet lakmitare pasi janë më pak të kushtueshme në kohë. Nëse dështojnë, provojmë programimin dinamik.

# 2. Activity Selection Problem

**Problemi i zgjedhjes së aktiviteteve** (në vazhdim *Activity Selection Problem*) është problem i optimizimit që merret me zgjedhjen e aktiviteteve jo-konfliktuale që duhet të ekzekutohen nga një person ose makinë, në një interval kohor të caktuar.

Formalisht, nëse kemi një bashkësi të **n** aktiviteteve që kanë kohën e fillimit dhe mbarimit, duhet të zgjedhim **numrin maksimal** të aktiviteteve jo-konfliktuale që mund të kryhen nga një person ose makinë, duke pasur parasysh se mund të merren me vetëm një aktivitet në një interval të caktuar kohor.

Supozojmë se kemi bashkësinë me aktivitete që duan të shfrytëzojnë një burim. Ai burim mund t’i shërbejë vetëm një aktiviteti në kohë të caktuar. Secili aktivitet ka **kohën e fillimit** dhe **kohën e mbarimit** ,ku . Nëse zgjedhet, aktiviteti zhvillohet në intervalin kohor gjysmë të hapur .

Tek Activity Selection Problem, dëshirojmë të zgjedhim nënbashkësinë më të madhe me aktivitete të pajtueshme reciprocikisht, pra që janë kompatibile me njëra-tjetrën, nuk bien ndesh në intervalin e caktuar kohor.

Aktivitetet dhe janë **kompatibile**, pra mund të ndodhin ose ekzekutohen së bashku pa pasur konflikt, nëse intervalet dhe nuk kanë pjesë të përbashkët. D.m.th. dhe janë kompatibile nëse ose .

Për fund, supozojmë se aktivitetet renditen në bazë të kohës së mbarimit në rend rritës.

Për shembull, marrim parasysh bashkësinë e mëposhtme të aktiviteteve:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|  | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 8 | 8 | 2 | 12 |
|  | 4 | 5 | 6 | 7 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |

Nënbashkësia përbëhet nga aktivitetet e pajtueshme reciprocikisht. Por të njëjtën përbërje ka edhe nënbashkësia ose . Algoritimet na ndihmojnë të gjejmë një nënbashkësi të tillë me numrin më të madh të mundshëm të elementeve.

## 2.1 Qasja me programimin dinamik

Shënojmë me bashkësinë e aktiviteteve që fillojnë pasi që aktiviteti ka mbaruar dhe

mbarojnë para se të fillojë aktiviteti .

Supozojmë se dëshirojmë të gjejmë bashkësinë më të madhe të aktiviteteve të pajtueshme reciprocikisht në . Supozojmë se bashkësia e tillë maksimale është , me elemente . Duke e përfshirë në zgjidhje optimale, mbetemi me **dy** nënprobleme: përcaktimi i aktiviteteve të pajtueshme reciprocikisht që i takojnë (aktivitetet që fillojnë pasi që ka mbaruar dhe mbarojnë para se të fillojë aktiviteti ) dhe (aktivitetet që fillojnë pasi që ka mbaruar dhe mbarojnë para se të fillojë aktiviteti ).

Le të jetë dhe , ashtu që përmbanë aktivitetet që mbarojnë para se të fillojë dhe gjenden në , ndërsa përmbanë aktivitetet që fillojnë para se të mbarojë dhe gjenden në .

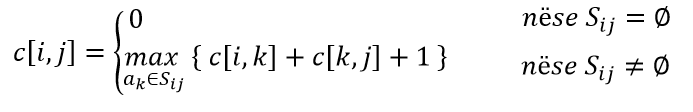
Kështu fitojmë . Si pasojë, , bashkësia maksimale që përmbanë aktivitetet e pajtueshme reciprocikisht nga , përbëhet nga aktivitete. Nga formula e fundit mund të vërejmë se **zgjidhja optimale** arrihet vetëm pasi që të zgjidhen dy nënproblemet dhe .

Nëse arrijmë të gjejmë një bashkësi të aktiviteteve të pajtueshme reciprocikisht që gjenden në , ku , atëherë mund të përdorim në vend të , në zgjidhje të nënproblemit për . Kështu do të formonim bashkësinë me elemente, që kontradikton supozimin se është zgjidhje optimale. Argument simetrik aplikohet edhe për aktivitetet që gjenden në .

Nëse shënojmë me madhësinë e zgjidhjes optimale të bashkësisë , fitohet rekurenca:

.

Sigurisht, nëse nuk do ta dinim se zgjidhja optimale e bashkësisë  e përfshinë aktivitetin , ne do të duhej të ekzaminonim të gjitha aktivitet në për të caktuar cilin aktivitet ta zgjedhim, kështu që:



Kjo mënyrë e karakterizimit të nënstrukturës optimale të problemit tregon se Activity Selection Problem mund të zgjidhet përmes programimit dinamik. Tani thjesht mund të zhvillojmë një algoritëm rekurziv dhe e memoizojmë, ose mund të punojmë poshtë-lartë dhe ta mbushim tabelën gjatë rrugës. Por do ta anashkalonim një karakteristikë shumë të rëndësishme të Activity Selection Problem që mund ta përdorim në avantazhin tonë.

## 2.2 Qasja me algoritmin lakmitar

Çka nëse do të mund të shtonim një aktivitet tek zgjidhja optimale pa pasur nevojë fillimisht të zgjidhim të gjitha nënproblemet? Kjo mund të na shpëtojë nga marrja në konsideratë e të gjitha mundësive për zgjedhje që trashëgohen nga rekurenca. Në fakt, për Activity Selection Problem, mund të konsiderojmë vetëm një zgjedhje: zgjedhjen lakmitare.

Çka nënkuptojmë me zgjedhje lakmitare? Intuitivisht kuptohet se duhet të zgjedhim një aktivitet që lejon burimin të ekzekutojë sa më shumë aktivitete tjera që është e mundur.

Nga të gjitha aktivitetet që zgjedhim, një prej tyre duhet të jetë i pari që mbaron. Prandaj, intuita na thotë se ne duhet të zgjedhim aktivitetin në **S** me kohën më të shpejtë të mbarimit. Kjo do ta lejonte burimin të jetë i lirë për sa më shumë aktivitete pasuese që është e mundur. Nëse më shumë se një aktivitet në **S** ka kohën më të shpejtë të mbarimit, atëherë zgjedhja bëhet sipas dëshirës.

Me fjalë të tjera, nëse aktivitetet janë të renditura në rend rritës në bazë të kohës së mbarimit, zgjedhja e parë lakmitare do të jetë menjëherë aktiviteti , duke kursyer kështu në kompleksitetin kohor.

Nëse bëjmë zgjedhje lakmitare, na mbetet vetëm **një** nënproblem për t’u zgjidhur: përcaktimi i aktiviteteve që fillojnë pasi që mbaron. Pse nuk duhet të konsiderojmë si tek programimi dinamik edhe përcaktimin e aktiviteteve që mbarojnë pasi që fillon? Sepse kemi dhe është koha më e hershme e mbarimit, prandaj nuk mund të ketë ndonjë aktivitet me kohë të mbarimit më të vogël ose barabartë me . Prandaj, të gjitha aktivitetet që janë kompatibile me  duhet të fillojnë pasi që mbaron.

Formalisht, tek pika **2.1** është vërtetuar se Activity Selection Problem ka nënstrukturë optimale. Po ashtu, le të jetë bashkësia e aktiviteteve që fillojnë pasi që aktiviteti mbaron. Nëse me zgjedhje lakmitare përcaktohemi për , atëherë mbetet problemi i vetëm për t’u zgjidhur.

Një pytetje e madhe mund të shtrohet. A është e saktë intuita jonë? A është gjithmonë zgjedhja jonë lakmitare, pra aktiviteti me kohën më të hershme të mbarimit, pjesë e një zgjidhjeje optimale? Teorema në vijim vërteton që kjo është e vërtetë.

**Teorema 1**:

Konsiderojmë çfarëdo nënproblemi jo të zbrazët dhe le të jetë një aktivitet në me kohën më të hershme të mbarimit. Atëherë përfshihet në nënbashkësinë maksimale të aktiviteteve të pajtueshme reciprocikisht të .

**Vërtetim**:

Le të jetë një nënbashkësi maksimale e aktiviteteve të pajtueshme reciprocikisht në . Le të jetë aktivitet në me kohën më të hershme të mbarimit.

Nëse , atëherë prova përfundon, sepse kemi treguar se gjendet në nënbashkësinë maksimale të aktiviteteve të pajtueshme reciprocikisht të .

Nëse , atëherë le të jetë bashkësia duke zëvendësuar me . Aktivitetet në janë disjunkte, sepse edhe aktivitetet në janë disjunkte. është aktiviteti i parë në që mbaron dhe . Pasi që , përfundojmë se është nënbashkësi maksimale e aktiviteteve të pajtueshme reciprocikisht e dhe përfshinë aktivitetin .

Kështu, e shohim se ndonëse mundemi të zgjidhim Activity Selection Problem me programim dinamik, nuk kemi nevojë. Për më tepër, nuk kemi pasur nevojë të ekzaminojmë fare nëse nënproblemet e Activity Selection Problem kanë pjesë të përbashkëta. Në vend të kësaj, vazhdimisht zgjedhim akitivitetin që mbaron i pari, i mbajmë vetëm aktivitetet kompatibile me këtë aktivitet, dhe përsërim të njëjtën gjë derisa nuk mbetet asnjë aktivitet.

Veç kësaj, pasi që gjithmonë zgjedhim aktivitetin me kohën më të hershme të mbarimit, kohët e mbarimit të aktiviteteve që zgjedhim duhet të jenë gjithsesi në rend rritës. Çdo aktivitet konsiderohet vetëm një herë, nëse janë të renditur në rend rritës sipas kohës së mbarimit.

Algoritmi që zgjidh Activity Selection Problem nuk ka nevojë të punojë poshtë-lartë, sikur algoritimi i bazuar në tabel i programimit dinamik. Në vend të kësaj, mund të punojë lartë-poshtë, duke vendosur një aktivitet në zgjidhje optimale dhe pastaj duke zgjidhur nënproblemin e zgjedhjes së aktiviteteve që janë kompatibil me atë që është zgjedhur më parë. Algoritmet lakmitare pikërishtë kanë këtë dizajn lartë-poshtë: bë një zgjedhje dhe pastaj zgjidh nënproblemin, në vend të teknikës poshtë-lartë që zgjidh nënproblemet para se të bëjë një zgjedhje.

## 2.3 Algoritmi lakmitar rekurziv

Tani që kemi parë se si të anashkalojmë qasjen e programimit dinamik dhe në vend të tij të përdorim algoritëm lakmitar, mund të shkruajmë drejtpërdrejtë një procedurë rekurzive për të zgjidhur Activity Selection Problem.

Funksioni **RECURSIVE-ACTIVITY-SELECTOR** merr si parametra kohën e fillimit dhe mbarimit të aktiviteteve, të përfaqësura nga vektorët dhe , indeksi që definon nënproblemin që do të zgjidh dhe madhësia e problemit origjinal. Si rezultat rikthen nënbashkësinë maksimale të aktiviteteve të pajtueshme reciprocikisht të . Supozojmë se aktivitetet e dhëna si vlera hyrëse janë të renditura sipas kohës së mbarimit në rend rritës. Nëse nuk janë të renditura, atëherë duhet ta kryejmë edhe këtë veprim. Me qëllim që algoritmi të fillojë punën, e shtojmë një aktivitet fiktiv me , kështu që nënproblemi është bashkësia e të gjitha aktiviteteve . Thirrja fillestare, që e zgjidh tërë problemin është *RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n)*.

// Gjej aktivitetin e parë në për mbarim

Në thirrjen e parë *RECURSIVE-ACTIVITY-SELECTOR (s, f, k, n)*, unaza **while** kërkon për aktivitetin e parë në për të mbaruar. Unaza ekzaminon , deri sa të gjejë aktivitetin e parë që është kompatibil me . Aktiviteti i tillë duhet të plotësojë kushtin . Nëse unaza përfundon ekzekutimin sepse ka gjetur një aktivitet të tillë, si rezultat kthehet unioni i me nënbashkësinë maksimale të , që fitohet nga thirrja rekurzive *RECURSIVE-ACTIVITY-SELECTOR (s, f, m, n).* Në mënyrë alternative, unaza mund të terminohet kur , rast në të cilin janë ekzaminuar të gjitha aktivitetet në  pa gjetur ndonjë kompatibil me . Kështu, prandaj dhe procedura kthen . Nga gjithë thirrjet rekurzive që realizohen, secili aktivitet testohet vetëm një herë në testin e unazës **while**. Në veçanti, aktiviteti ekzaminohet krejt në fund kur .

## 2.4 Algoritmi lakmitar iterativ

Lehtësisht, procedurën rekurzive të pikës **2.3** mund ta konvertojmë në procedurë iterative. Procedura *RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)* është pothuajse “tail recursive”: përfundon me thirrje rekurizve të vetes i ndjekur nga operacioni union. Është e zakonshme të ndodh transformimi i dretëpërdrejtë nga forma tail-rekurzive në atë iterative; në fakt, disa kompajler për gjuhë programuese të caktuara e kryejnë këtë detyrë automatikisht.

Procedura **GREEDY-ACTIVITY-SELECTOR** është versioni iterativ i *RECURSIVE-ACTIVITY-SELECTOR*. Supozon se aktivitetet janë të renditura në rend rritës sipas kohës së mbarimit. I ruan aktivitetet e zgjedhura në një bashkësi dhe e rikthen bashkësinë si rezultat.

Procedura funksionon si në vijim. Variabla indekson aktivitetin e fundit të shtuar në . Pasi që aktivitet i konsiderojmë të renditura sipas kohës së mbarimit në rend rritës, është gjithmonë maksimumi i kohëve të mbarimit të cilitdo aktivitet në . D.m.th. .

Fillimisht zgjedhet aktiviteti , inicializohet që të përmbajë vetëm këtë aktivitet dhe inicializohet që të indeksojë këtë aktivitet.

Në unazën **for** gjendet aktiviteti i me kohën më hershme të mbarimit. Unaza kontrollon secili aktivitet dhe e shton atë në bashkësinë nëse është kompatibil me aktivitet e zgjedhura më parë.

Për të vërtetuar që aktiviteti është kompatibil me aktivitetet e zgjedhura më parë, kontrollohet nëse koha e tij e fillimit, , nuk është më e hershme se koha e mbarimit, , e aktivitetit më të fundit të vendosur në .

Nëse aktiviteti është kompatibil, atëherë vendoset në bashkësinë dhe tani e tutje duhet të indeksojë këtë element të ri të shtuar në bashkësi, si elementi i fundit.

## 2.5 Kompleksiteti kohor

Në shkencë kompjuterike, kompleksiteti kohor përshkruan sasinë e kohës që do të kalojë një algoritëm gjatë ekzekutimit. Kompleksiteti kohor i një algoritmi kuantifikion sasinë e kohës që merr algoritmi të ekzekutohet në funksion të gjatësisë së inputit.

Në analizën e algoritmeve, Big O përdoret shpesh për të përshkruar rastin më të keq të një algoritmi, duke marrur termin me shkallën më të lartë të një funksioni polinomial, duke injoruar të gjitha vlerat konstante pasi që nuk kanë shumë ndikim në rezultat në rastet kur inputi është i madh.

Për Activity Selection Problem kemi përmendur dy qasje të zgjidhjes: programimin dinamik dhe algoritmin lakmitar.

Nëse zgjidhja realizohet me **programim dinamik**, detyrohemi të përdorim dy unaza. Në këtë rast, kompleksiteti kohor është sepse algortimi do të detyrohet të kryejë operacione.

Nëse zgjidhja realizohet me **algoritmin lakmitar** kemi raste të ndryshme të kompleksitetit kohor. Duke i paraprocesur të dhënat hyrëse ose duke përdor një strukturë të përshtatshme të të dhënave (zakonisht një priority queue), zgjedhjet lakmitare mund t’i bëjmë shumë më shpejtë, duke bërë kështu algoritmin më efikas. Rastet më lartë i kemi trajtuar nën supozimin se aktivitetet janë të renditura në bazë të kohës së mbarimit në rend rritës dhe ky paraprocesim i inputit na mundëson që secilin aktivitet ta ekzaminojmë vetëm njëherë. Në rast se nuk do të kishte sortim, atëherë ndryshon efikasiteti i zgjedhjes së aktiviteteve kompatibile.

Në rastin kur aktivitet janë të sortuar, tek algoritimi lakmitar, kompleksiteti i kohës është . Pasi që algoritmi pranon input me madhësi , do t’i kryejë operacione. Pasi përdoret vetëm një unazë, në mënyrë lineare bëjmë kërkimin për një element. Prandaj kompleksiteti kohor është ose linear sepse unaza ka cikle.

Në rastin kurelementet nuk janë të sortuar, tek algoritimi lakmitar, kompleksiteti i kohës është **,** për shkak të kompleksitetit të sortimit.

Fillimisht, paraqitet kur arrijmë tek rezultati duke iteruar vetëm në disa elemente të një bashkësie të të dhënave, në vend të iterimit në gjithë elementet. Por, kur kjo përsëritet herë, atëherë kompleksiteti kohor arrin në .

Në implementim të kodit vërehet versioni iterativ i algoritimi lakmitar pa sortim paraprak të inputit.

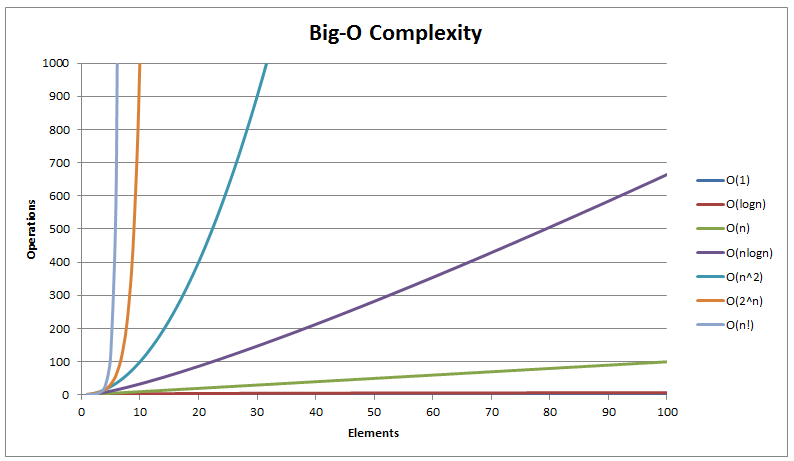


Figure - Big O Notation

## 2.6 Aplikimi i Activity Selection Problem

Ka shumë shembuj nga jeta reale ku përdoren algoritmet lakmitare (Greedy Algorithms). P.sh, merrni parasysh problemin e shndërrimit të një numri arbitrar të centëve në monedha standarde (euro), me fjalë të tjera, merrni parasysh problemin e kthimit të kusurit. Procesi që pothuajse me siguri ndiqni, pa e marrë parasysh me vetëdije, së pari është që të ktheni kusurin duke përdorur numrin më të madh të eurove (p.sh 500, 200, 100, 50, 20, 10, 5, 2, 1) e pastaj të centëve (p.sh 50, 20, 10, 5). Ky është një shembull i të punuarit me lakmi: në secilin hap, ne zgjodhëm përfitimin maksimal të menjëhershëm (numrin e monedhave që mund të jepnim).

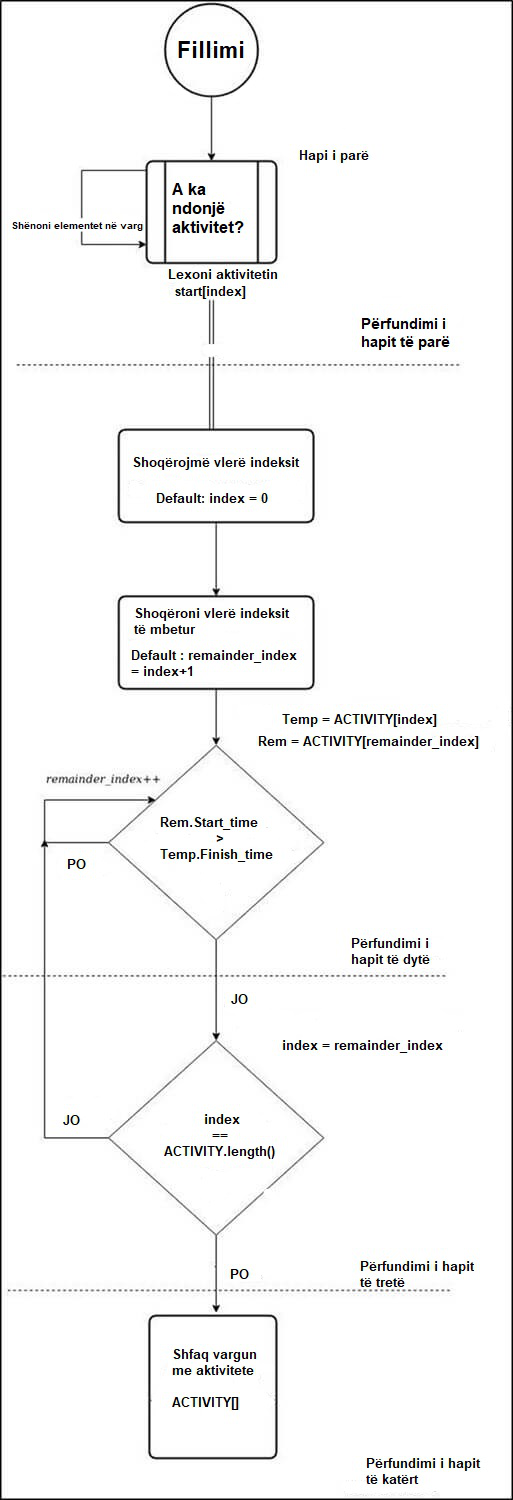
Studentët dhe nxënësit nëpër teste zakonisht kur zgjidhin problemet, shikojnë vështirësitë e pytjeve duke kalkuluar përafërsisht kohën që do ju merr për të përgjigjur, në mënyrë që të arrijnë të shënojnë sa më shumë përgjigje duke lënë në fund ato pytje të cilat kërkojnë më shumë kohë.

Një menaxher i projektit gjatë ndarjes së detyrave përgjatë ditës për punëtorin e tij tenton që në orarin e task-ave të vendosë sa më shumë task-a të cilat nuk janë të varura nga koha e përfundimit të ndonjë task-e nga ndonjë punëtorë tjetër, në mënyrë që të mos ketë kohë të pashfrytëzuar dhe të kryhet sa më shumë punë.



Figure - Ilustrim i zgjedhjeve lakmitare

# 3. Bllok-diagrami i algoritmit



# 4. Implementimi

Algoritimi i implementuar në gjuhën programuese Python:

|  |
| --- |
| def Activities(s, f):     n = len(f)      i = 0     replace = df[0]["Resource"] = "Active"      for j in range(n):         if Time(s[j], f[i]):              replace = df[j]["Resource"] = "Active"              i = j |

Rezultati që fitohet pas ekzekutimi të gjithë programit të zhvilluar:



Figure - Selektimi i aktiviteteve

Me ngjyrë të kuqe janë aktivitet të cilat nuk mund të zhvillohen dhe nuk janë kompatibile me aktivitetet tjera që janë dalluar me ngjyrë të gjelbërt.

# Referencat

1) Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, Clifford Stein. Introduction to   
 Algorithms. The MIT Press Cambridge, Massachusetts London, England.

2) [Activity Selection Problem](https://www.studytonight.com/data-structures/activity-selection-problem)

3) [Activity Selection Problem using Greedy algorithm](https://iq.opengenus.org/activity-selection-problem-greedy-algorithm/)

4) [Some common runtime complexities and their meanings](https://medium.com/learn-with-the-lean-programmer/some-common-runtime-complexities-and-their-meanings-5a2bf4320f48)

5) [Activity Selection Problem | Greedy Algorithm](https://www.codesdope.com/course/algorithms-activity-selection/)

6) [Activity Selection Problem using Dynamic Programming](https://www.techiedelight.com/activity-selection-problem-using-dynamic-programming/)

7) [Activity Selection Problem | Greedy Algo-1](https://www.geeksforgeeks.org/activity-selection-problem-greedy-algo-1/)

8) [An Activity Selection Problem](https://www.javatpoint.com/activity-selection-problem)

9) [Activity Selection Problem](https://www.tutorialspoint.com/Activity-Selection-Problem)

10) [Introduction to Greedy Algorithms | GeeksforGeeks](https://www.youtube.com/watch?v=HzeK7g8cD0Y)