**1.Naloga:** Recimo, da imamo dan seznam A[1 ... n] števil, ki so lahko pozitivna, negativna ali nič. Števila niso nujno cela števila.

1. Opišite in analizirajte algoritem, ki najde največjo vsoto elementov v podseznamu seznama A[1 ... n]
2. Opišite in analizirajte algoritem, ki najde največji zmnožek elementov v podseznamu seznama A[1 ... n]

**Problem (največja vsota podseznama (maximum subarray sum)):**

Ta problem je prvi zastavil Ulf Grenander leta 1977 kot poenostavljen model za oceno največje verjetnosti vzorcev v digitaliziranih slikah.

Tega problema bi se lahko lotili tako, da bi poiskali vse možne podsezname danega seznama in zanje izračunali vsoto. Na koncu bi vrnili največjo izmed poračunanih.

Recimo, da smo na mestu i. Potem pogledamo vse podsezname od i naprej, kot kaže slika. Na vsakem podseznamu poračunamo vsoto, ki jo primerjamo s prej dobljeno vsoto.

**Skica ideje algoritma:**

A diagram of a mathematical system

Description automatically generated with medium confidence

**Koda:**

A computer screen shot of a code

Description automatically generated

Ta algoritem je časovno zelo potraten . Da se ga pa narediti precej hitrejšega. S tem se je ukvarjal Joseph Born Kadane, ki je ugotovil učinkovit algoritem. Ta se imenuje Kadane's Algorithm in sodi v poglavje dinamičnega programiranja. Probleme v tem poglavju se je potrebno lotiti tako, da upoštevamo tri napotke in sicer:

* Rešimo problem na manjšem podproblemu,
* Shranjujmo rešitve manjših podproblemov,
* Ponovno uporabimo shranjeno rešitev podproblema, namesto da ponovno poračunavamo rešitev.

Tak način zagotavlja večjo hitrost.

Preden opišemo Kadane-ov algoritem si poglejmo nekaj posebnosti:

* Če seznam vsebuje samo pozitivna števila, je odgovor trivialen in sicer največja vsota je kar vsota celega seznama.
* Če seznam vsebuje samo negativna števila, je odgovor 0.
* Več različnih podseznamov ima lahko enako največjo vsoto.

**Ideja Kadanovega algoritma:**

Ideja je imeti dve spremenljivki. Ena, ki hrani največjo vsoto podseznama do trenutnega indeksa (recimo ji trenutnaVsota) in drugo, ki hrani največjo vsoto podseznamov najdenih do sedaj (recimo ji maxVsota). Torej trenutnaVsota opredelimo kot maksimalno vsoto tistih podzaporedij, ki se končajo na indeksu, kjer smo. Ko smo na koraku i, vemo kakšna je trenutnaVsota podzaporedij, ki se končajo na i-1. Ko pa gledamo i-ti element, novo tranutnaVsota naračunamo tako, da:

* Če je element pozitiven, bo trenutnaVsota stara povečana za element na i-tem mestu.
* Če je element negativen, pa imamo dve možnosti:
  + Element se nam morda splača vzeti, ker bo služil kot 'most' do nadaljnjih novih maksimalnih trenutnihVsot. To se splača vzeti, če stara trenutnaVsota s to prišteto negativno vrednostjo ne pade pod 0.
  + Če povečana vsota pade pod 0, pa se nam elementa ne splača vzeti. Bolje je vzeti podzaporedje na i+1 mestu, zato trenutnaVsota nastavimo na 0.

Hkrati preverjamo tudi največjo vsoto do sedaj (maxVsoto), ki je lahko sedaj trenutnaVsota ali pa ostane maxVsota od prej.

**Skica ideje algoritma:**

A whiteboard with writing on it

Description automatically generated

**Koda:**

A computer code with text

Description automatically generated with medium confidence

**Analiza časovne in prostorske zahtevnosti:**

* Velikost problema: velikost seznama
* Karakteristična operacija: seštevanje in primerjanje velikosti
* Časovna zahtevnost: , ker iteriramo samo enkrat čez seznam
* Prostorska zahtevnost: , ker uporabljamo samo dve spremenljivki in nič dodatnega prostora npr. dodatnega seznama.

**Primeri:**

**A white board with colorful text

Description automatically generated with medium confidence**

**A math equation with numbers and symbols

Description automatically generated with medium confidence**

**Viri za a):**

<https://en.wikipedia.org/wiki/Maximum_subarray_problem>

<https://algodaily.com/lessons/kadanes-algorithm-explained>

<https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/>

**Problem (največji produkt podseznama (maximum subarray product):**

Ideja problema je enaka kot za naivni način pri vsoti. Prav tako je tudi enaka časovna zahtevnost, zato se bomo raje osredotočili na uporabo Kadanovega algoritma na tem primeru.

**Ideja algoritma:**

Na mestu i imamo od prej poračunan največji in najmanjši produkt podseznama. Sedaj imamo dve možnosti za nov največji produkt. Lahko je to vrednost na mestu i ali pa je vrednost prejšnjega podseznama z največjim zmnožkom pomnoženega, z vrednostjo na mestu i. To označujemo z lokalni\_max. Enak razmislek je za najmanjšo vrednost produkta podseznama, ki ga označujemo z lokalni\_min. Pri seštevanju nimamo 'problema' s predznaki števila, tukaj pa lahko naletimo na tri možnosti, v primeru, da je število na mestu i negativno in sicer:

* Naj bosta lokalni\_max > 0 in lokalni\_min < 0. Ko ju pomnožimo z vrednostjo na mestu i, bo lokalni\_max \* i < 0 in lokalni\_min \* i > 0. Sedaj bo nov lokalni\_min večji od novega lokalnega\_max.
* Naj bosta lokalni\_max > 0 in lokalni\_min > 0. Ko ju pomnožimo z vrednostjo na mestu i, bo lokalni\_max \* i < 0 in lokalni\_min \* i < 0. Sedaj bo nov lokalni\_min večji od novega lokalnega\_max.
* Naj bosta lokalni\_max < 0 in lokalni\_min < 0. Ko ju pomnožimo z vrednostjo na mestu i, bo lokalni\_max \* i > 0 in lokalni\_min \* i > 0. Sedaj bo nov lokalni\_min večji od novega lokalnega\_max.

Od tod ideja, da ko naletimo na negativno vrednost, zamenjamo lokalni\_min in lokalni\_max.

**Skica ideje algoritem**:

A white paper with text and numbers

Description automatically generated

**Koda:**

A screen shot of a computer code

Description automatically generated

**Analiza časovne in prostorske zahtevnosti:**

* Velikost problema: velikost seznama
* Karakteristična operacija: množenje in primerjanje velikosti
* Časovna zahtevnost: , ker iteriramo samo enkrat čez seznam
* Prostorska zahtevnost: , ker uporabljamo samo tri spremenljivki in nič dodatnega prostora npr. dodatnega seznama.

**Primeri:**

A whiteboard with text on it

Description automatically generated

A page of a math problem

Description automatically generated with medium confidence

**Viri za b):**

<https://www.youtube.com/watch?v=bLHrFVx-OJQ>

<https://takeuforward.org/data-structure/maximum-product-subarray-in-an-array/>

<https://algodaily.com/lessons/kadanes-algorithm-explained>

**2.Naloga:** Naloga od nas zahteva, da izračunamo, koliko časa bi potrebovali za izračun najkrajše poti med dvema ogliščema u in v, v tako imenovanem 'looped tree' z n vozlišči, z uporabo Dijkstrovega algoritma. Dijkstrov algoritem je algoritem za iskanje najkrajših poti med vozlišči v uteženem grafu. Primer tega bi bil problem razvoza iz optimizacije.

**Delovanje Dijkstrovega algoritma:**

Ideja Dijkstrovega algoritma je najti najkrajšo pot med začetno točko in vsemi drugimi točkami v grafu. Algoritem vzdržuje množico točk, kjer je za vsako točko shranjena trenutna najkrajša znana pot od vira do te točke. Postopek poteka iterativno, pri čemer se v vsakem koraku izbere točka, ki ni v množici, vendar ima najmanjšo trenutno znano razdaljo od vira. Nato se posodobi razdalja do vseh sosednjih točk izbrane točke. Postopek se ponavlja, dokler niso vse točke vključene v množico ali pa ni več možnih izboljšav poti. Torej naš končen rezultat bo drevo najcenejših povezav.

Vendar v naši nalogi imamo opravka s potjo med izbranima vozliščema. Originalni Dijkstrov algoritem potem dopolnimo tako, da vedno začnemo v izbranem vozlišču u in končamo postopek, ko najdemo vozlišče v.

Potrebovali bomo:

* Seznam razdalj, ki hrani najmanjšo razdaljo od izvornega do vsakega vozlišča v grafu. Na začetku je za izbrano vozlišče razdalja enaka 0, za vse ostale pa neskončno. Čim bomo našli krajšo razdaljo, se bo seznam ustrezno posodobil.
* Vrsta , ki je prednostna vrsta vseh vozlišč v grafu. Na koncu bo prazna.
* Niz , ki označuje, katera vozlišča je algoritem že obiskal. Na koncu bo vseboval vsa vozlišča.

**Primer:**

Želeli bi najkrajšo pot od vozlišča 4 do vozlišča 8.

1. Postopek začnemo v vozlišču 4.
2. Pogledamo povezave iz njega in izračunamo povezave za sosednja vozlišča.
3. Naslednje vozlišče izberemo tako, da ima najkrajšo povezavo. To je 1.

Postopek nadaljujemo.

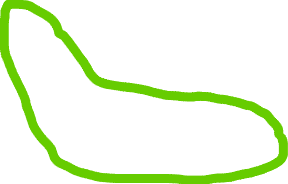
A diagram of a network

Description automatically generated

Naše drevo sedaj izgleda kot označeno na sliki. Izberemo tisto vozlišče, ki ima najkrajšo povezavo. Te povezave so zapisane z rdečo in obkrožene s temno zeleno. Vidimo, da je to ravno vozlišče 2.

A diagram of a network

Description automatically generated



Potem ga dodamo v naše drevo in ponovimo postopek, dokler ne pridemo do vozlišča 8.

A diagram of a network

Description automatically generated

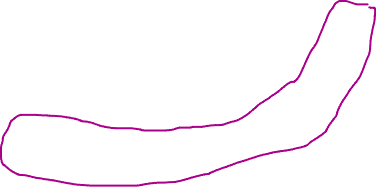


Sedaj smo prišli do vozlišča 8 in postopek končali. Dobili smo drevo najcenejših povezav.

Ampak nas zanima samo od vozlišča 4 do vozlišča 8, kar je pa del naše cele poti, zato je rezultat obarvan vijolično.

A diagram of a network

Description automatically generated



A diagram of a diagram

Description automatically generated

A diagram of a triangle with lines and numbers

Description automatically generated with medium confidence

A diagram of a triangle with circles and lines

Description automatically generated

**Psevdo koda Dijakstrovega algoritma:**

**def** dijakstra(graf = (,), izbrani):

dist[izbrani] ← 0

postavi dist[] = 0 za vse in ≠ izbrani

dodaj v

←

**while**  **≠** :

← vozlišče v z min(dist[])

odstrani

dodaj v

za vse sosede od dodanega vozlišča , ki še niso v drevesu najkrajših

poti popravimo dist[]

**return** dist[]

**Analiza časovne in prostorske zahtevnosti za Dijakstrov algoritem:**

Naj bo velikost vozlišča in velikost povezave/cene.

Časovna zahtevnost je odvisna od izbire podatkovne strukture za . Če je:

* je običajna vrsta:
* je prednostna vrsta v obliki kopice:
* je prednostna vrsta v obliki Fibonaccijeve kopice: .

Najhitrejši algoritem je z uporabo Fibonaccijeve kopice.

Sedaj vemo, kako deluje Dijakstrov algoritem, vendar mi imamo posebno vrsto drevesa in to je 'looped tree'.

A diagram of a tree

Description automatically generated

Opazimo lahko, da ima vsako vozlišče največ dve povezavi, ki gresta iz njega. Torej v najslabšem primeru, če gremo skozi vse povezave, gremo dvakrat skozi vsa vozlišča. Časovna zahtevnost za povezave je , saj je . Skupna časovna zahtevnost za Dijakstrov algoritem je .

**Opiši in analiziraj boljši algoritem za ta problem.**

Če pogledamo sliko lahko ugotovimo, da obstaja samo eno vozlišče, ki ima več kot eno povezavo, ki kaže vanj. To je koren. Najkrajša pot med vozliščema in bo lahko šla čez koren ali pa ne. Poglejmo si oba primera.

1. Vozlišči in ne gresta čez koren. Rešitev je tako samo ena in sicer seštevek cen povezav od do . To lahko opravimo v .

A diagram of a tree

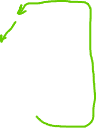
Description automatically generated



1. Vozlišči in gresta čez koren. Od korena do vozlišča bo pot enolična in jo lahko naračunamo enako kot v primeru a). Potrebujemo izračunati le najkrajšo pot od do korena. Od do korena pa imamo usmerjeni acikličen graf, torej graf, ki je usmerjen in brez ciklov. Zanj prav tako obstaja algoritem za računanje najkrajših poti, ki ima časovno zahtevnost Skupen čas za algoritem je torej

A diagram of a tree

Description automatically generated



**Primeri:**

**A drawing of a diagram

Description automatically generated**

A drawing of a diagram

Description automatically generated

A drawing of a diagram

Description automatically generated

**A drawing of a diagram

Description automatically generated**

**A diagram of a tree

Description automatically generated with medium confidence**

**A diagram of a tree

Description automatically generated**

**Viri:**

<https://www.baeldung.com/cs/dijkstra-time-complexity>

<https://www.baeldung.com/cs/priority-queue>

<https://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/>

<https://tildesites.bowdoin.edu/~ltoma/teaching/cs231/duke_cps130/Homeworks/Solutions/H21-solution.pdf>

<https://www.youtube.com/watch?app=desktop&v=71Z-Jpnm3D4>