Praktično delo 2

Tine Fajfar

April 21, 2021

1 Opazovanje časa izvajanje glede na pričakovani čas

Najprej si ogledamo čase izvajanja za posamezen algoritem in razložimo pričakovane oziroma nepričakovane rezultate. Za vsak algoritem razmislimo kako parametri vplivajo na čas izvajanja in posledično kako bi morali izbrati parametre ter hkrati generirati števila, da bi se približali zgornji meji časovne zahtevnosti algoritmov. Pri približnih algoritmih opišemo tudi način, kako doseči čim slabši rezultat, ki ga vrne algoritem.

Pri analizi sem generiral probleme velikosti $n = [100, 100000]$, $n = n \cdot 1.25$ ter $k = [100, 1000000]$, $k = k \cdot 1.25$. Za algoritem FPTAS sem generiral še $\varepsilon = [n/100, n]$, $\varepsilon = \varepsilon + n/100$. Pri tem sem opazoval čase izvajanja posameznega algoritma, pri čemer sem čas izvajanja reševanja posameznega problema omejil na 30 sekund.

Pomembno je poudariti, da sem v programskem jeziku Java števila generiral naključno z uporabo razreda `Random` na intervalu $[0, k)$.

1.1 Dinamično programiranje

Pri dinamičnem programiranju na sliki 1 opazimo skoraj popolnoma linearno premico časa izvajanja glede na pričakovani čas izvajanja. To je pričakovano, saj moramo ne glede na števila vedno preiskati celoten prostor $n \times k$. To pomeni, da ne glede na vrednosti vedno opravimo enako število primerjanj, branj ter pisanj iz oziroma v pomnilnik.

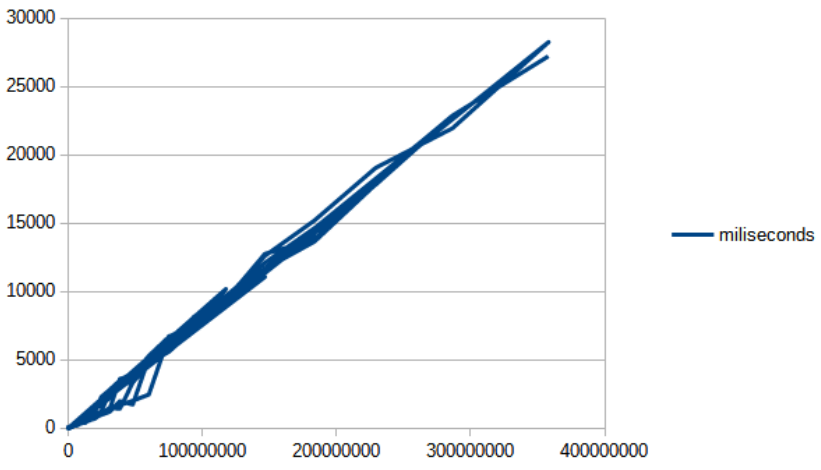

Figure 1: x os predstavlja pričakovan čas izvajanja, na y osi je čas izvajanja v milisekundah

1.2 Izčrpno preiskovanje

Pri izčrpnem preiskovanju na sliki 2 vidimo, da je situacija precej bolj zanimiva. Res je pričakovan čas $O(2^n)$, vendar na čas izvajanja v praksi pomembno vpliva tudi k, sploh ker smo ga uporabili tudi kot zgornjo mejo intervala, iz katerega jemljemo naključna števila. Namreč manjši kot je k bolj je verjetno, da seštevek poljubnih števil preseže k in ga zato zavržemo. Prav tako, ko jemljemo števila iz manjšega intervala, povečamo možnost, da sta števili enaki oziroma je seštevek dveh števil enak nekemu številu, ki ga že imamo v seznamu, torej ga bomo zavrgli.

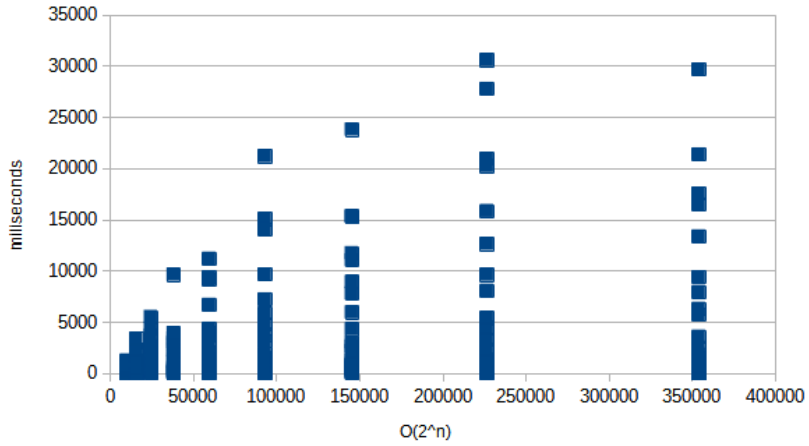

Figure 2: Pričakovani čas izvajanja izčrpnega preiskovanje proti dejanskemu času izvajanja v milisekundah.

Da bo izvajalni čas algoritma kar najslabši, moramo ohranjati naš seznam čim daljši. To storimo tako, da najprej zagotovimo dovolj velik k oziroma dovolj majhna števila, da ga le-ta nikoli ne presežejo. Poleg tega zagotovimo, da se števila seštejejo vedno v novo število. To lahko naredimo tako, da števila generiramo v naslednjem zaporedju: $L = 2^0, 2^1, 2^2, ..., 2^{(n-1)}$. Da naša ideja deluje v praksi, prikazuje slika 3.

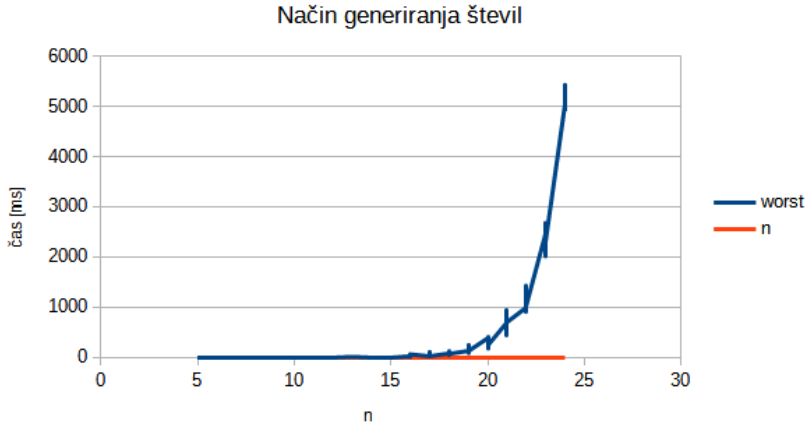

Figure 3: Primerjava izvajalnega časa pri naključnem generiranju števil proti najslabšemu primeru pri parametrih $n = [5, 24]$; $k = 2^{30}$.

1.3 Požrešen algoritem

Pri požrešnem algoritmu na sliki 5 najprej opazimo, da so časi izvajanja tudi za zelo velike instance zelo majhni. V primerjavi s točnima algoritmoma so krajši tudi za faktor 600. Največ časa vzame urejanje seznama elementov, ki predstavlja kar celotno časovno zahtevnost algoritma, saj se moramo nato le enkrat sprehoditi čez seznam elementov, da dobimo rešitev. Predpostavljam, da na čas izvajanja najbolj vpliva urejenost generiranih števil, saj to najznatnejše vpliva na čas urejanja elementov. Elemente hranim v `ArrayList`, uredim pa jih s klicem metode `Collections.sort`, ki interno uporablja *mergesort*. Torej moramo za najdaljši čas izvajanja poizkati najslabši primer instance za mergesort.

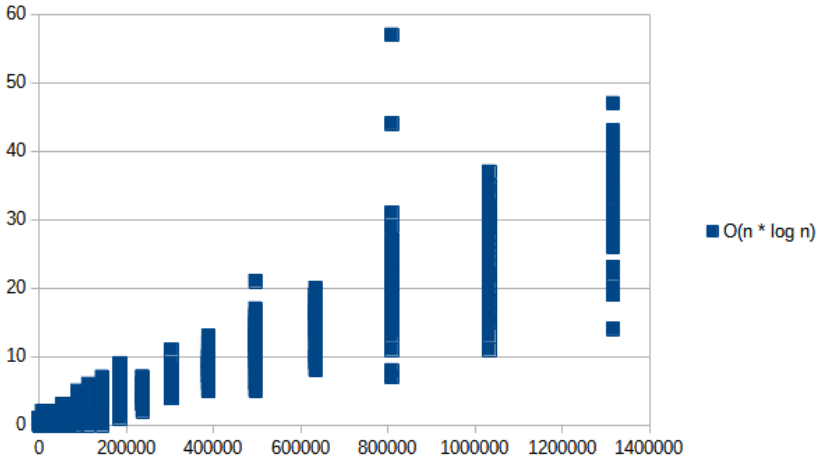

Figure 4: Pričakovani čas izvajanja izčrpnega preiskovanje proti dejanskemu času izvajanja v milisekundah.

Kako dobiti najslabši rezultat pa smo si pogledali že na vajah. Največje število oziroma seštevek prvih x_i števil v padajoče urejenem seznamu, ki predstavlja največje število, mora biti od zgoraj čim bliže meji $k/2$, pri tem pa ne sme obstajati nobeno drugo tako število, da bi ga lahko dodali v množico rešitve.

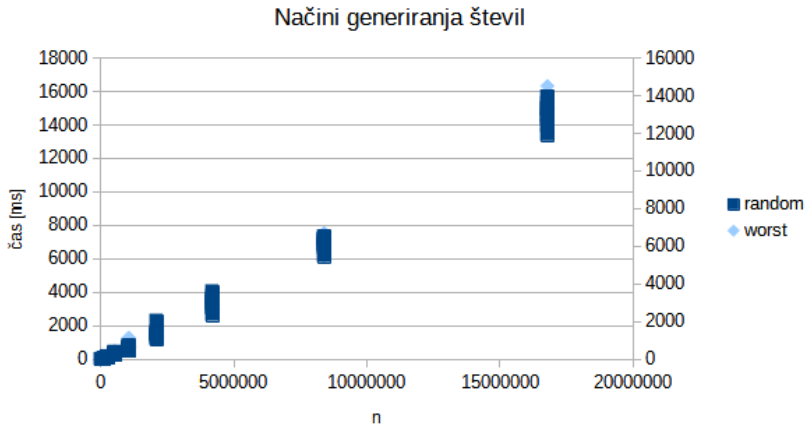

Figure 5: Primerjava časa izvajanja pri naključnem in načrtnem generiranju števil za parametra: $n = [2^{10}, 2^{24}]$, $k = n$. Opaznih razlik ni.

1.4 FPTAS

Za algoritem FPTAS sem na sliki 6 vključil le izvajalne čase za algoritem pri fiksnem $\varepsilon = 1$. Na sliki vidimo več vrhov kar je pričakovano, saj sem v tem eksperimentu uporabil zares ekstremne vrednosti, ki jih pričakovan asimptotični čas izvajanja ne opiše dobro. Namreč v skrajnem primeru bi lahko imeli zelo velik pričakovan čas izvajanja zaradi ogromnega k in hkrati $|n| = 1$ kar bi pomenilo, da bi se algoritem vseeno izvedel v O(1), saj bi morali pogledati le, ali število je v množici ali ni.

V splošnem pa za poljuben n in k generiramo časovno najslabši primer tako, da ohranjamo naš seznam čim daljši. Prvi pogoj je dovolj majhen ε, da nikoli ne izvedemo rezanja - dejansko s tem FPTAS spremenimo v požrešni algoritem iz točke 2, ki teče v času $O(2^n)$. Drugi in tretji pogoj sta nato enaka kot pri izčrpnem preiskovanju v poglavju 1.2.

Poljubno slab rezultat algoritma dosežemo precej preprosto. Na prvo mesto v seznam elementov postavimo poljubno majhno število, nato pa izberemo $\varepsilon >> 2n$ s čimer odrežemo vsa nadaljna števila, ki jih dobimo v naslednjih korakih.

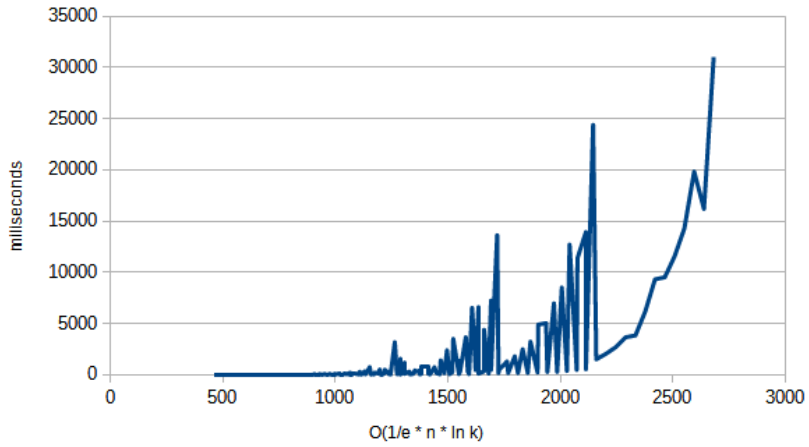

Figure 6: Pričakovani čas izvajanja izčrpnega preiskovanje proti dejanskemu času izvajanja v milisekundah.

2 Vpliv ε na čas izvajanja algoritma FPTAS

Najprej smo generirali vhodni seznam števil, ki je bil vedno isti. Algoritem smo nato izvedli 250-krat z naslednjimi parametri: $n = 1000, k = 100000, \varepsilon = [1, 250]$. Na sliki 7 je lepo vidna pričakovana eksponentna rast časa izvajanja z manjšanjem ε.

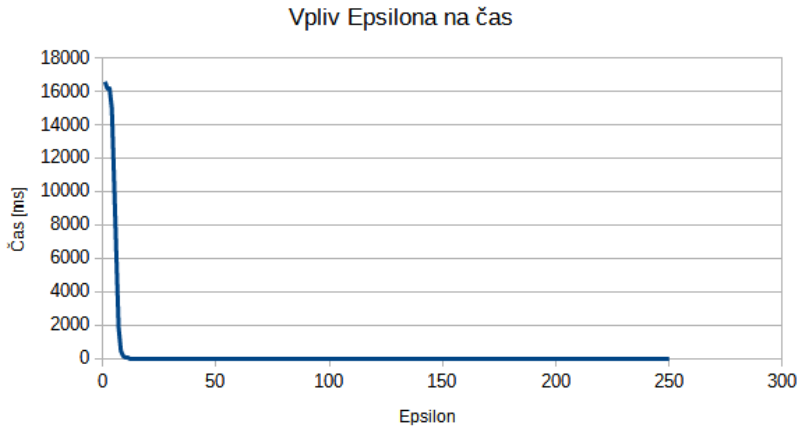

Figure 7: Pričakovan eksponentni padec izvajalnega časa algoritma FPTAS z večanjem ε. $n = 1000, k = 100000$