**Universiteti për Biznes dhe Teknologji**

**Master Shkenca Kompjuterike dhe Inxhinieri**



**Top K Frequent Elements**

**Lënda: Dizajnimi dhe Analizimi i Algoritmeve**

Profesoreshë: Punoi:

Dr. Sc. Ermira Daka Elma Redzepi

Prishtinë, 2025

**Top K Frequent Elements**

Elma Rexhepi – [er242575418@ubt-uni.net](mailto:er242575418@ubt-uni.net)

University for Business and Technology, 10000 Prishtine, Kosovo

**Abstrakti:** Ne studiojmë problemin e gjetjes së k artikujve më të shpeshtë në një rrjedhë artikujsh për masën e frekuencës maksimale të propozuar së fundmi. Bazuar në vetitë e një artikulli, frekuenca maksimale e një artikulli llogaritet në një dritare rrëshqitëse gjatësia e së cilës ndryshon në mënyrë dinamike. Identifikimi i elementeve më të shpeshtë në një koleksion të dhënash është një problem i zakonshëm në shkencën e të dhënave dhe inteligjencën artificiale.Përveç të qenit pa parametra, kjo mënyrë e matjes së mbështetjes së artikujve u tregua se kishte avantazhin e një zbulimi më të shpejtë të shpërthimeve në një rrjedhë, veçanërisht nëse grupi i artikujve është heterogjen. Algoritmi "Top K Frequent Elements" synon të gjejë K elementët me shpeshtësinë më të lartë në një listë të dhënë. Kjo problematikë është e rëndësishme në aplikacione të ndryshme, përfshirë analizën e të dhënave, indeksimin e dokumenteve dhe sistemet e rekomandimeve. Algoritmi që u propozua për mirëmbajtjen e të gjithë artikujve të shpeshtë, megjithatë, shkallëzohet dobët kur numri i artikujve bëhet i madh. Prandaj, në këtë punim ne propozojmë, në vend që të raportojmë të gjithë artikujt e shpeshtë, të minojmë vetëm ato top-k më të shpeshtat. Së pari vërtetojmë se për të zgjidhur saktësisht këtë problem, ne kemi ende nevojë për një sasi të kufizuar memorie (të paktën lineare në numrin e artikujve). Ku ne kete punim eksplorohen qasjet e ndryshme për zgjidhjen e këtij problemi, përfshirë përdorimin e strukturave të të dhënave si **heap** dhe **hashmap**, si dhe metodat më efikase për të trajtuar të dhëna të mëdha. Megjithatë, në disa kushte të arsyeshme, ne tregojmë teorikisht dhe empirikisht se ekziston një algoritëm efikas në memorie. Është implementuar një prototip i këtij algoritmi dhe ne paraqesim performancën e tij, efikasiteti i kujtesës në të dhënat e jetës reale dhe në eksperimentet e kontrolluara me të dhëna sintetike. Në fund, analizohen kompleksiteti i algoritmeve dhe rastet e tyre të përdorimit në situata reale.

**PËRMBAJTJA**

[**Lista e figurave** 4](#_Toc194009770)

[**1.** **HYRJE** 5](#_Toc194009771)

[**2.** **TOP K FREQUENT ELEMENTS** 6](#_Toc194009772)

[**3.** **DIZAJNIMI DHE PËRZGJEDHJA E ALGORITMIT - ALGORITMI(I) I ZGJEDHUR, PSEUDOKODI, JUSTIFIKIMI** 7](#_Toc194009773)

[**3.1. Metoda 1: Përdorimi i një "Hash Map" dhe "Sorting" (O(N log N))** 7](#_Toc194009774)

[3.1.1. CFG (Control Flow Graph) – Hash-Map 9](#_Toc194009775)

[**3.2. Metoda 2: Përdorimi i "Min-Heap" (O(N log K))** 10](#_Toc194009776)

[3.2.1. CFG (Control Flow Graph) – Min-Heap 12](#_Toc194009777)

[**3.3. Metoda 3: Përdorimi i "Bucket Sort" (O(N))** 14](#_Toc194009778)

[3.3.1. CFG (Control Flow Graph) – Bucket-Sort 16](#_Toc194009779)

[**3.4. Metoda 4: Përdorimi i "Priority Queue" (O(N))** 18](#_Toc194009780)

[3.4.1. CFG (Control Flow Graph) – Priority Queue 20](#_Toc194009781)

[**4.** **ANALIZA E KOMPLEKSITETIT KOHOR/HAPËSINOR, DIZAJNI I RAFINUAR** 21](#_Toc194009782)

[**4.1. "Hash Map" dhe "Sorting" (O (N log N))** 21](#_Toc194009783)

[**4.2. Min-Heap (O (N log K))** 22](#_Toc194009784)

[**4.3.** **Bucket Sort (O(N))** 22](#_Toc194009785)

[**4.4.** **Priority Queue (O (N log N))** 23](#_Toc194009786)

[**4.5.** **Krahasimi metodave -- HashMap – Min-Heap – Bucket-Sort dhe Priority Queue** 24](#_Toc194009787)

[**5.** **ZBATIMI I KODIT, RASTE BAZË TË TESTIMIT** 25](#_Toc194009788)

[**6.** **ANALIZA DHE EKSPERIMENTIMI I PERFORMANCËS SË ALGORITMEVE PËR GJETJEN E K ELEMENTEVE MË TË SHPESHTA** 33](#_Toc194009789)

[**7.** **PËRFUNDIMI** 37](#_Toc194009790)

[**8.** **REFERENCAT** 38](#_Toc194009791)

# **Lista e figurave**

[Figure 1. HashMap, Heap dhe Output-i 6](#_Toc194009745)

[Figure 2. CFG HasMap 9](#_Toc194009746)

[Figure 3. CFG Insertion Min-Heap 12](#_Toc194009747)

[Figure 4. Hapi 1: Insertimi Bucket Sort 16](#_Toc194009748)

[Figure 5. Hapi 2: Insertimi Bucket Sort 16](#_Toc194009749)

[Figure 6. Hapi 3: Insertimi Bucket Sort 16](#_Toc194009750)

[Figure 7. Hapi 4: Insertimi Bucket Sort 17](#_Toc194009751)

[Figure 8. Hapi 5: Rezultati pas Insertim-it 17](#_Toc194009752)

[Figure 9. CFG Priority Queue 20](#_Toc194009753)

[Figure 10. Krahasimi i algoritmeve me 60 elemente 33](#_Toc194009754)

[Figure 11. Krahasimi i algoritmeve në sekonda të testimit të performancës 34](#_Toc194009755)

[Figure 12. Krahasimi i algoritmeve në Dataseta 34](#_Toc194009756)

[Figure 13. Krahasimi i algoritmeve (HashMap, Heap, Bucket Sort dhe Priority Queue) në grafike 35](#_Toc194009757)

[Figure 14. Krahasimi i algoritmeve (HashMap, Heap, Bucket Sort) në grafike 35](#_Toc194009758)

[Figure 15. Krahasimi i algoritmeve (HashMap, Heap, Bucket Sort dhe Priority Queue) në grafike 36](#_Toc194009759)

# **HYRJE**

Në epokën e informacionit, analizimi i të dhënave të mëdha është bërë një sfidë thelbësore për fusha të ndryshme, si inteligjenca artificiale, tregtia elektronike dhe kërkimi në internet. Një nga problemet më të zakonshme është gjetja e elementeve më të shpeshtë brenda një grupi të madh të dhënash. Kjo detyrë është kritike në shumë aplikacione praktike, për shembull, për të identifikuar fjalët më të përdorura në një tekst, produktet më të shitur në një dyqan online, ose kërkimet më të zakonshme në një motor kërkimi.

Problemi "Top K Frequent Elements" mund të përshkruhet si më poshtë: duke pasur një listë të numrave (ose fjalëve, ose ndonjë objekti tjetër), kërkohet të gjenden K elementët që shfaqen më shpesh në atë listë. Një qasje naive për këtë problem do të ishte numërimi i shfaqjeve të secilit element dhe renditja e tyre sipas frekuencës. Megjithatë, për të dhëna të mëdha, qasjet naive mund të jenë joefikase dhe të kërkojnë shumë burime përpunimi.

Gjithashtu, ky punim analizon kompleksitetin kohor dhe hapësinor të këtyre metodave dhe aplikacionet e tyre në situata të ndryshme reale. Numërimi dhe identifikimi i artikujve që ndodhin shpesh, ose "goditësit e rëndë", është një nga më të rëndësishmet dhe metrika intuitive për të fituar njohuri mbi të dhënat në shkallë të gjerë. Mënyra naive për të nxjerrë artikujt top-K nga a rrjedha e të dhënave është për të numëruar numrin e saktë të dukurive të çdo artikulli të veçantë, pastaj rendit histogramin për të marrë artikujt më të shpeshtë. Kjo qasje naive por popullore vuan nga një kompleksitet kohor prej **O (n log n)**, në të cilin n është numri i përgjithshëm i elementeve në grupin e të dhënave, dhe gjithashtu një kërkesë për hapësirë e **O(n)**, duke supozuar se renditja kryhet në hapësirë ​​lineare. Në një mjedis të shpërndarë, ku të dhënat copëtimi është i zakonshëm, problemi është mjaft i rëndë. Duhet të mbajmë një histogram të frekuencës lokale në secilën nyje, e cila zakonisht është me madhësi n vetë. Këto histograme lokale do të duhet të komunikohen nëpër nyje, dhe pasuar nga operacionet globale të bashkimit dhe renditjes. Kështu, çdo nyje do të kishte nevojë për të komunikuar histograme me madhësi O(n), të cilat mund të çojnë në një pengesë të rëndësishme komunikimi.

Ky punim shqyrton metodat më efikase për të gjetur K elementët më të shpeshtë, duke përfshirë:

* Përdorimin e një **hashmap** për numërimin e frekuencës së elementeve
* Implementimin e një **heap-i** (priority queue) për të mbajtur K elementët më të shpeshtë
* Qasjen e renditjes dhe zgjedhjes së pjesëshme për përmirësim të efikasitetit

# **TOP K FREQUENT ELEMENTS**

Ky projekt fokusohet në zhvillimin dhe analizën e një algoritmi për gjetjen e **K elementeve më të shpeshta** në një grup të dhënash. Një nga problemet klasike të përpunimit të të dhënave është se ka aplikime në analiza të mëdha të të dhënave.

**Qëllimet kryesore të projektit janë:**

* Zhvillimi i një algoritmi efikas për gjetjen e K elementeve më të shpeshta në një grup të dhënash të dhënë.
* Krahasimi i metodave të ndryshme për zgjidhjen e problemit, duke përfshirë përdorimin e **Heap**, **HashMap**, dhe **Bucket Sort**.
* Analiza e kompleksitetit kohor dhe hapësinor të algoritmeve për të përcaktuar metodën më të mirë për skenarë të ndryshëm.
* Implementimi i zgjidhjes në **Python**, me përdorimin e strukturave të të dhënave si **priority queue (heapq)**.
* Vlerësimi i performancës së algoritmit duke përdorur grupe të dhënash reale ose të gjeneruara.
* **Libraritë Kryesore:** heapq, collections, dhe numpy për krijimin e të dhënave.
* **Të dhëna për testim:** Të dhëna sintetike ose të dhëna reale nga burime publike (p.sh. fjalët më të përdorura në një tekst të madh).
* **Metoda të vlerësimit:** Matje e kohës së ekzekutimit për dataset të madhësive të ndryshme dhe krahasimi i saktësisë së metodave.

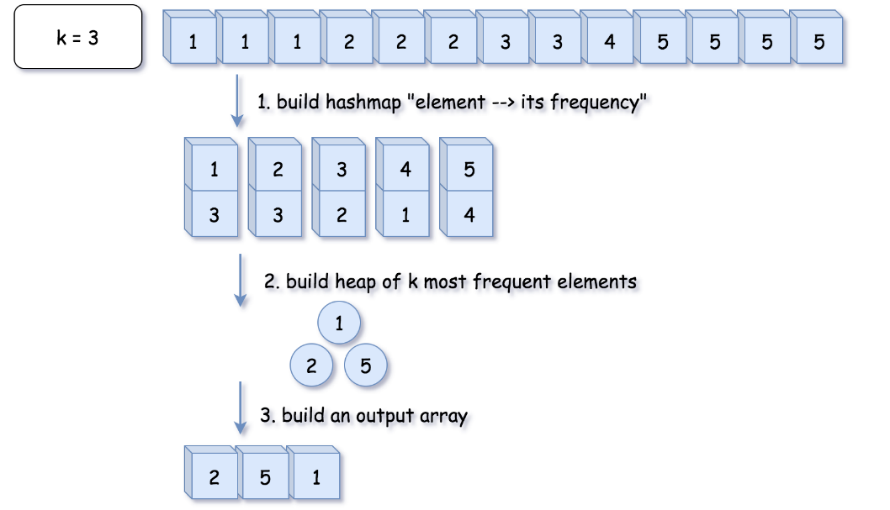


Figure 1. HashMap, Heap dhe Output-i

# **DIZAJNIMI DHE PËRZGJEDHJA E ALGORITMIT - ALGORITMI I ZGJEDHUR, PSEUDOKODI, JUSTIFIKIMI**

"Top K Frequent Elements" është një problem klasik në analizën e të dhënave dhe algoritmet, ku kërkohet të gjenden **K elementët më të shpeshtë** në një listë ose një varg numrash. Ky problem është i rëndësishëm në fusha si **përpunimi i të dhënave, inteligjenca artificiale, kërkimi në internet dhe analiza e teksteve**.

**Ky problem përdoret në shumë aplikacione praktike, përfshirë:**

* **Analiza e të dhënave:** Gjetja e produkteve më të shikuara nga përdoruesit.
* **Makinat e rekomandimeve:** Identifikimi i artikujve më të blerë ose të lexuar.
* **Analiza e teksteve:** Gjetja e fjalëve më të përdorura në një dokument (p.sh., në një analizë të fjalëve kyçe në SEO).
* **Sistemet e caching:** Ruajtja e të dhënave më të përdorura në memorie për performancë më të mirë.
* **Siguria kibernetike:** Identifikimi i adresave IP më të zakonshme në sulmet kibernetike.

**Si përdoret për të zgjidhur këtë problem?**

Ka disa mënyra për të zgjidhur këtë problem në mënyrë efikase. Metodat më të zakonshme janë:

## **3.1. Metoda 1: Përdorimi i një "Hash Map" (O(N log N))**

* Përdorim një **HashMap** për të numëruar frekuencën e secilit element.
* Numëron shpeshtësinë e secilit element duke përdorur një "Hash Map" (Dictionary në Python).
* Sortimi i elementeve sipas shpeshtësisë.
* Marrja e K elementëve më të shpeshtë.

**Pseudokodi për "Hash Map" (O(N log N))**

FUNCTION TopKFrequentSorting(nums, k):

*// Hap 1: Krijo një hartë për numërimin e frekuencave*

frequency\_map ← empty map

FOR each num IN nums:

IF num EXISTS in frequency\_map THEN

frequency\_map[num] ← frequency\_map[num] + 1

ELSE

frequency\_map[num] ← 1

*// Hap 2: Konverto hartën në një listë çiftesh (numër, frekuencë)*

frequency\_list ← list of (num, frequency) from frequency\_map

*// Hap 3: Rendit listën sipas frekuencës në mënyrë zbritëse*

SORT frequency\_list BY frequency DESCENDING

*// Hap 4: Merr k elementët e parë nga lista e renditur*

result ← empty list

FOR i FROM 0 TO k-1:

APPEND frequency\_list[i][0] TO result

RETURN result

Analiza e strukturimit t**ë** HashMap-it:

∑ᵢ₌₁ⁿ 1 = n → **O(n)**

**Sqarim:**

* **Së pari, ndërtojmë një hartë të frekuencave.**
* Përdorim një **Hash Map (Dictionary)** për të numëruar sa herë paraqitet secili numër në listë.
* **Më pas, konvertojmë Hash Map-in në një listë me çifte (numër, frekuencë).**
* Kjo listë na ndihmon të ruajmë shpeshtësitë në një format që mund të renditet.
* **Rendisim listën sipas frekuencës në mënyrë zbritëse.**
* Sortimi bëhet duke krahasuar frekuencat e secilit element.
* **Marrim K elementët e parë nga lista e renditur.**
* Këta janë numrat që shfaqen më shpesh në listë.

**Hash Map** për llogaritjen e shpeshtësisë është **O(N).**

**Hash Map** na ndihmon të gjejmë shpeshtësitë në **O(N)**.

**Sortimi i listës** pastaj kërkon **O(N log N)**.

**Marrja e K elementëve** merr **O(K)**, por kjo është e shpejtë për K të vogël.

### 3.1.1. CFG (Control Flow Graph) – Hash-Map

Tani do të shohim një CFG (Control Flow Graph) të Hash-Map-it se si funksionin më detajisht përmes CFG-së.

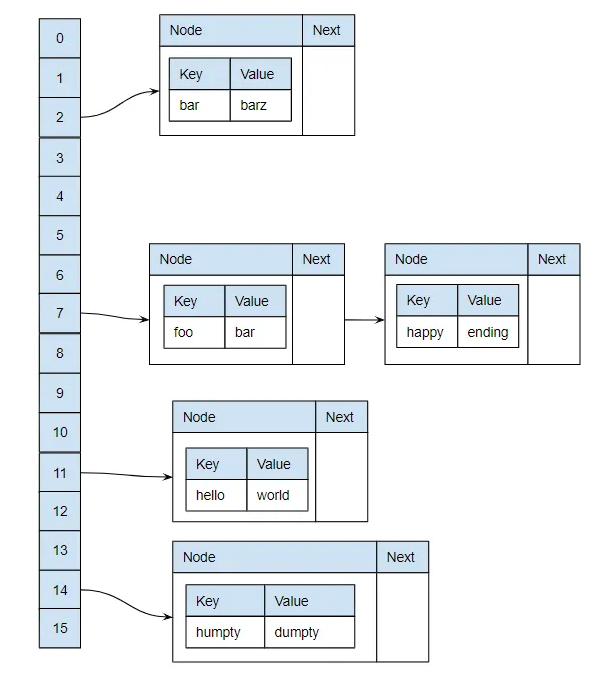


Figure 2. CFG HasMap

* **Vizualimi se si bëhet për HashMap kemi linkun, ku do të shohim në mënyrë më të kjartë përdorimin e algoritmit HashMap:** **http://hashmap-visualization.vercel.app/**

Pra duke u bazuar në CFG më lartë tani do të japim edhe disa sqarime për krahasimin e Hash-Map në integer dhe stringe, se si vijojn dhe cila është më optimale.

**Krahasimi i HashMap me tipe të ndryshme të të dhënave (Integer vs. String)**

HashMap në Java (dhe struktura të ngjashme në Python si dict) përdor funksionin hashCode () për të llogaritur indeksin ku duhet të ruhet një çelës. Ky funksion ka kosto të ndryshme për Integer dhe String.

**1. HashMap me Integer**

* **Funksion hash**: Për Integer, hashCode() kthen vetë vlerën e numrit (p.sh., 10.hashCode() == 10).
* **Koha e ekzekutimit**: Është shumë e shpejtë, sepse operacioni hash është O (1).
* **Përplasje (Collisions)**: Shumë të pakta, sepse integer-at janë unike.

**2. HashMap me String**

**Funksion hash**: Për String, hashCode() bën një kombinim të karaktereve duke përdorur një formulë:

S [0] × 31 (n−1) + s [1] × 31 (n−2) + ... + s [n−1] s [0] \ times 31^{(n-1)} + s [1] \ times 31^{(n-2)} + ... + s [n-1] s [0] × 31 (n−1) + s [1] × 31(n−2) + ... + s [n−1]

**Koha e ekzekutimit**: O(n), ku n është gjatësia e vargut.

**Përplasje (Collisions)**: Më të mundshme se tek Integer, veçanërisht për fjalë të ngjashme.

**Optimizim**: Java bën perturbim hash (h = key.hashCode() ^ (h >>> 16)) për të reduktuar përplasjet.

**Krahasimi i Performancës:**

|  |  |  |
| --- | --- | --- |
| Tipi i të dhënave | Koha e hashCode() | Rreziku i përplasjeve |
| Integer | O(1) | Shumë i ulët |
| String | O(n) | Mund të ndodhë për fjalë të ngjashme |

**Konkluzion:** **HashMap me Integer është më e shpejtë**, sepse hashCode() është O(1). HashMap me String ka **kosto më të lartë**, veçanërisht për fjalë të gjata.

**Shembull Python:**

hashmap = {}

*# Integer si çelës*

hashmap[10] = "Value for 10"

*# String si çelës*

hashmap["hello"] = "Value for hello"

print(hashmap[10]) *# Shpejtë (O(1))*

print(hashmap["hello"]) *# Pak më i ngadaltë për shkak të llogaritjes së hash-it.*

## **3.2. Metoda 2: Përdorimi i "Min-Heap" (O(N log K))**

Kjo metodë përdor një Heap me prioritet për të mbajtur vetëm K elementët më të shpeshtë në memorie.

* Numëron shpeshtësinë e secilit element me një Hash Map.
* Shton elementet në një Min-Heap (ku ruajmë vetëm K elementët më të shpeshtë).
* Nxjerrja e rezultateve nga heap-i.

**Pseudokodi për "Min-Heap" (O(N log K))**

FUNCTION TopKFrequentMinHeap(nums, k):

*// Hap 1: Krijo një hartë për numërimin e frekuencave*

*frequency\_map ← empty map*

FOR each num IN nums:

IF num EXISTS in frequency\_map THEN

frequency\_map[num] ← frequency\_map[num] + 1

ELSE

frequency\_map[num] ← 1

*// Hap 2: Përdor një Min-Heap për të ruajtur K elementët më të shpeshtë*

*min\_heap ← empty min heap*

FOR each (num, frequency) IN frequency\_map:

INSERT (frequency, num) INTO min\_heap

IF SIZE of min\_heap > k THEN

REMOVE min element from min\_heap

*// Hap 3: Nxjerr të gjithë elementët nga heap-i dhe i vendos në listën e rezultatit*

*result ← empty list*

WHILE min\_heap is NOT empty:

APPEND EXTRACT\_MIN(min\_heap)[1] TO result

RETURN result

**Analiza e strukturimit** **të Min-Heap-it:**

∑ᵢ₌₁ⁿ log i ≈ n → **O(n)**

**Sqarim:**

* **Së pari, ndërtojmë një hartë të frekuencave.**
* Përdorim një **Hash Map** për të numëruar shpeshtësinë e secilit numër në listë.
* **Pastaj, përdorim një Min-Heap për të ruajtur vetëm K elementët më të shpeshtë.**
* Futim elementët në një **Min-Heap**, duke ruajtur (frekuencë, numër).
* Nëse madhësia e heap-it tejkalon K, e heqim elementin më pak të shpeshtë.
* **Pas kalimit nëpër të gjithë numrat, nxjerrim elementët nga heap-i.**
* Këta janë K elementët më të shpeshtë, sepse ruajtëm vetëm më të shpeshtët në heap.

**Min-Heap lejon shtimin dhe fshirjen e elementeve në O(log K).**

**Në fund, nxjerrja e elementeve merr O(K log K), por kjo është e vogël në krahasim me O(N log K).**

### 3.2.1. CFG (Control Flow Graph) – Min-Heap

Tani do të shohim një CFG (Control Flow Graph) të Min-Heap se si funksionin më detajisht përmes CFG-së.

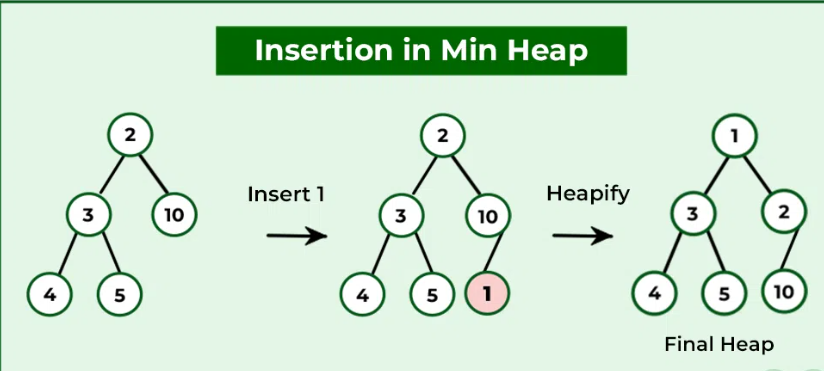


Figure 3. CFG Insertion Min-Heap

**Vizualimi se si bëhet për Min-Heap kemi linkun, ku do të shohim në mënyrë më të kjartë përdorimin e algoritmit Min-Heap: https://www.cs.usfca.edu/~galles/visualization/Heap.html**

**Futja në strukturën e të dhënave Min-Heap**

Operacioni i futjes në një grumbull të vogël përfshin hapat e mëposhtëm:

* Shtoni elementin e ri në fund të grumbullit, në pozicionin tjetër të disponueshëm në nivelin e fundit të pemës.
* Krahasoni elementin e ri me prindin e tij. Nëse prindi është më i madh se elementi i ri, ndërrojini ato.
* Përsëriteni hapin 2 derisa prindi të jetë më i vogël ose i barabartë me elementin e ri, ose derisa elementi i ri të arrijë rrënjën e pemës.
* Elementi i ri është tani në pozicionin e tij të saktë në grumbullin min dhe vetia e grumbullit është e plotësuar.

**Shembull Python:**

def insert\_min\_heap(heap, value):

*# Shtoni elementin e ri në fund të grumbullit (heap)*

heap.append(value)

*# Merrni indeksin e elementit të fundit*

index = len(heap) - 1

*# Krahasoni elementin e ri me prindin e tij dhe ndërroni vendet nëse është e nevojshme*

while index > 0 and heap[(index - 1) // 2] > heap[index]:

heap[index], heap[(index - 1) //

2] = heap[(index - 1) // 2], heap[index]

*# Lëvizni lart në pemë te prindi i elementit aktual*

index = (index - 1) // 2

*# Krijo një grumbull bosh (min-heap)*

heap = []

*# Lista e vlerave për t'u futur në grumbull*

values = [10, 7, 11, 5, 4, 13]

*# Fut çdo vlerë në grumbull dhe shfaq përmbajtjen e tij*

for value in values:

insert\_min\_heap(heap, value)

print(f"U fut {value} në min-heap: {heap}")

**Output:**

Inserted 10 into the min-heap: 10

Inserted 7 into the min-heap: 7 10

Inserted 11 into the min-heap: 7 10 11

Inserted 5 into the min-heap: 5 7 11 10

Inserted 4 into the min-heap: 4 5 11 10 7

**Krahasimi i performancës:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tipi i të dhënave | Shtimi (insert) | Heqja (remove) | Krahasimi | Përdorimi i memories |
| Integer | O(log N) | O(log N) | O(1) | I ulët |
| String | O(log N × n) | O(log N × n) | O(n) | I lartë |

**Shembull Python:**

import heapq

import time

*# Min-Heap me Integer*

heap\_int = []

values\_int = [10, 7, 11, 5, 4, 13]

start\_time = time.time()

for value in values\_int:

heapq.heappush(heap\_int, value)

end\_time = time.time()

print("Koha për Min-Heap me Integer:", end\_time - start\_time)

*# Min-Heap me String*

heap\_str = []

values\_str = ["apple", "banana", "cherry", "date", "fig", "grape"]

start\_time = time.time()

for value in values\_str:

heapq.heappush(heap\_str, value)

end\_time = time.time()

print("Koha për Min-Heap me String:", end\_time - start\_time)

* **Min-Heap me Integer është më i shpejtë dhe më efikas** për shkak të krahasimit O(1).
* **Min-Heap me String është më i ngadaltë**, sepse krahasimet kërkojnë O(n) operacione.

## **3.3. Metoda 3: Përdorimi i "Bucket Sort" (O(N))**

Përdoret kur kemi një gamë të vogël të vlerave. Ideja është të vendosim elementet në grupe të organizuara sipas shpeshtësisë së tyre.

* Numëron shpeshtësinë e secilit element me një Hash Map.
* Ruan elementet në "buckets" ku indeksi i listës tregon shpeshtësinë e një elementi.
* Lexon "bucket"-at nga fundi për të gjetur K elementët më të shpeshtë.

**Pseudokodi për "Bucket Sort" (O(N))**

FUNCTION TopKFrequentBucketSort(nums, k):

*// Hap 1: Krijo një hartë për numërimin e frekuencave*

frequency\_map ← empty map

FOR each num IN nums:

IF num EXISTS in frequency\_map THEN

frequency\_map[num] ← frequency\_map[num] + 1

ELSE

frequency\_map[num] ← 1

*// Hap 2: Gjej frekuencën maksimale për të krijuar "bucket"-at*

max\_frequency ← MAXIMUM value among all frequency\_map values

*// Hap 3: Krijo një listë me bucket-e; çdo bucket korrespondon me një frekuencë*

buckets ← list of (max\_frequency + 1) empty lists

*// Hap 4: Vendos secilin numër në bucket-in përkatës sipas frekuencës së tij*

FOR each (num, frequency) IN frequency\_map:

APPEND num TO buckets[frequency]

*// Hap 5: Fillimisht nga bucket me frekuencën më të madhe, grumbullo numrat derisa të mblidhen k elemente*

result ← empty list

FOR freq FROM max\_frequency DOWN TO 1:

FOR each num IN buckets[freq]:

APPEND num TO result

IF LENGTH(result) EQUALS k THEN

RETURN result

RETURN result

**Analiza e strukturimit** **të Bucket-sort:**

∑ᵢ₌₁ᵏ O(n/k \* log(n/k)) = n log(n/k) → **O(n)** (nëse k ≈ n)

**Sqarim:**

* Së pari ndërtojmë një hartë të frekuencave.
* Më pas, përcaktojmë frekuencën maksimale për të ndërtuar një listë (array) me bucket-e, ku indeksi i secilit bucket përfaqëson një frekuencë.
* Pastaj, çdo element vendoset në bucket-in përkatës bazuar në frekuencën e tij.
* Duke filluar nga bucket me frekuencën më të lartë, nxjerrim elementet derisa të arrijmë K elemente.

### 3.3.1. CFG (Control Flow Graph) – Bucket-Sort

Tani do të shohim një CFG (Control Flow Graph) të Bucket-Sort se si funksionin më detajisht përmes CFG-së.

**Hapi 1:**

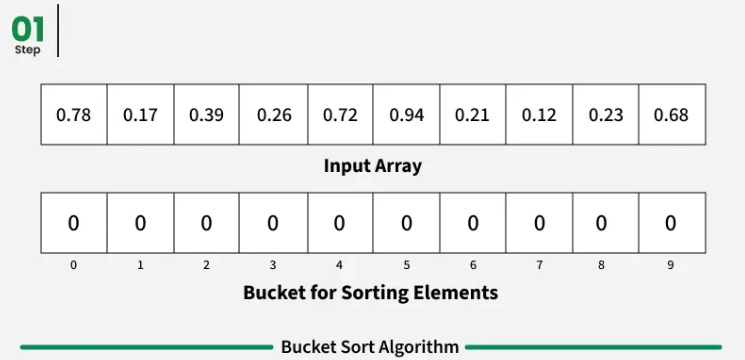


Figure 4. Hapi 1: Insertimi Bucket Sort

**Hapi 2:**

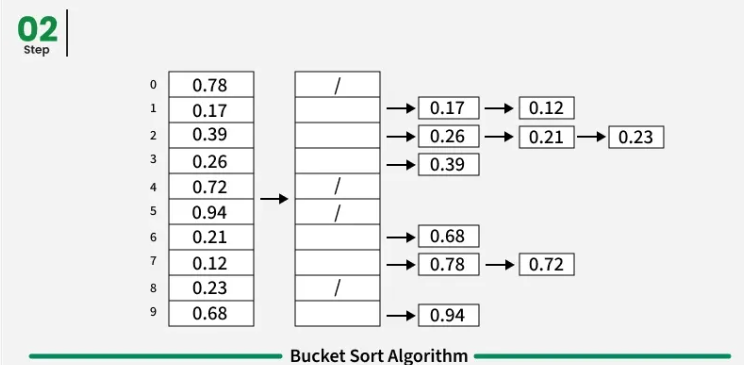


Figure 5. Hapi 2: Insertimi Bucket Sort

**Hapi 3:**

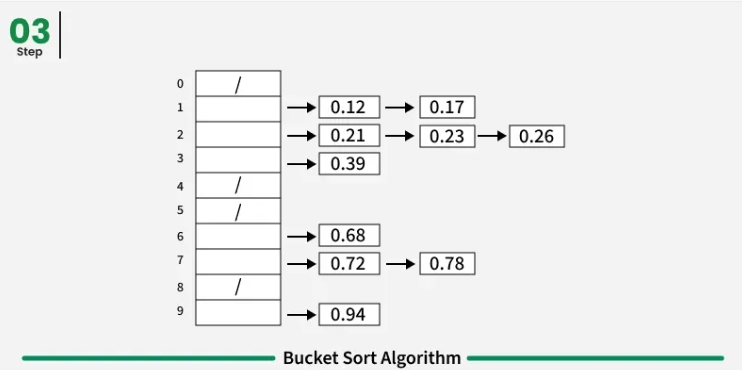


Figure 6. Hapi 3: Insertimi Bucket Sort

**Hapi 4:**

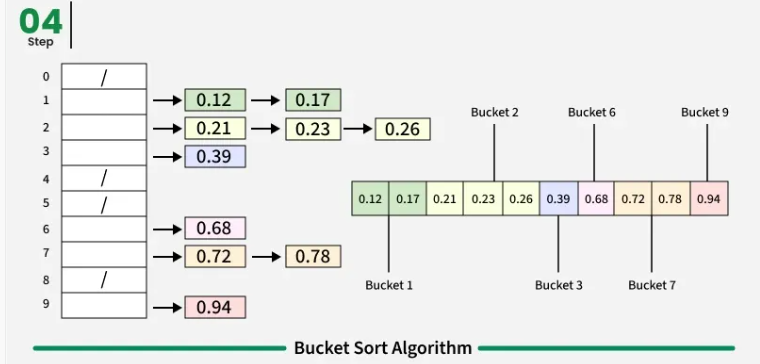


Figure 7. Hapi 4: Insertimi Bucket Sort

**Hapi 5: Rezultati final**

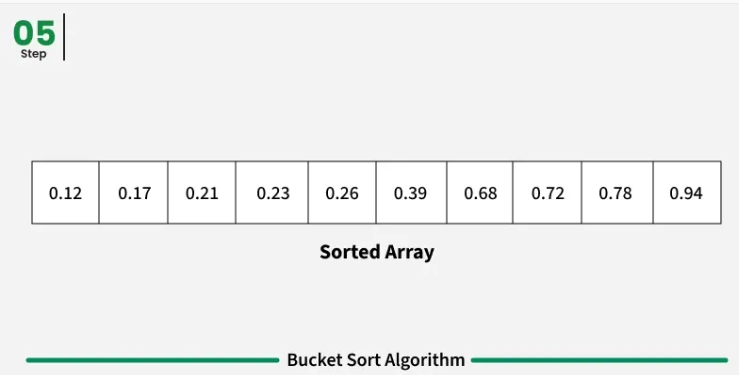


Figure 8. Hapi 5: Rezultati pas Insertim-it

**Vizualimi se si bëhet për Bucket-Sort kemi linkun, ku do të shohim në mënyrë më të kjartë përdorimin e algoritmit Bucket-Sort: https://www.cs.usfca.edu/~galles/visualization/BucketSort.html**

**Shembull Python:**

def bucket\_sort(arr):

if not arr:

return []

buckets = [[] for \_ in range(len(arr))] *# Krijojmë kovat (buckets)*

for num in arr:

index = int(num \* len(arr)) *# Përcaktojmë kovën përkatëse*

buckets[index].append(num)

for bucket in buckets:

bucket.sort() # Rendisim çdo kovë

return [num for bucket in buckets for num in bucket] *# Bashkojmë rezultatet*

*# Shembull përdorimi*

arr = [0.42, 0.32, 0.23, 0.52, 0.25, 0.47]

sorted\_arr = bucket\_sort(arr)

print(sorted\_arr)

* Krijon kovët (buckets) si lista bosh.
* Vendos elementet në kovën përkatëse bazuar në vlerën e tyre.
* Rendit çdo kovë veçmas me sort().
* Bashkon të gjitha kovët për të marrë listën e renditur.

**Krahasimi i performancës:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tipi i të dhënave | Vendosja në Kovë | Renditja e Kovës | Krahasimi | Përdorimi i memories |
| Integer | O(1) | O(k log k) | O(1) | I ulët |
| String | O(1) | O(k log k × m) | O(m) | I lartë |

* Bucket Sort është më efikas për integer, sepse krahasimi është O(1).
* Bucket Sort është më i ngadaltë për string, sepse krahasimet kërkojnë O(m) për secilin string.

## **3.4. Metoda 4: Përdorimi i "Priority Queue" (O(N))**

**Priority Queue** është një strukturë të dhënash e ngjashme me një radhë të zakonshme (queue), por me një veçori të veçantë: çdo element ka një **prioritet** dhe elementi me prioritetin më të lartë shërbehet i pari, pavarësisht nga rendi i futjes në radhë.

**Përdorimi i Priority Queue:** Për të ruajtur dhe nxjerrë elementët me frekuencën më të lartë në mënyrë efikase.

**Pseudokodi për “Priority Queue” O (N (log K))**

FUNCTION TopKFrequentBucketPriorityQueue(nums, k):

*// Hap 1: Ndërto një hartë për të numëruar frekuencën e secilit element*

frequency\_map ← empty map

FOR each num IN nums:

IF num EXISTS in frequency\_map THEN

frequency\_map[num] ← frequency\_map[num] + 1

ELSE

frequency\_map[num] ← 1

*// Hap 2: Gjej frekuencën maksimale*

max\_frequency ← MAXIMUM value among all frequency\_map values

*// Hap 3: Krijo një listë me "bucket"-e*

buckets ← list of (max\_frequency + 1) empty lists

*// Hap 4: Vendos secilin numër në bucket-in përkatës sipas frekuencës*

FOR each (num, frequency) IN frequency\_map:

APPEND num TO buckets[frequency]

*// Hap 5: Përdor një Priority Queue për të nxjerrë elementët më të shpeshtë*

pq ← empty max-heap (priority queue

FOR freq FROM max\_frequency DOWN TO 1:

FOR each num IN buckets[freq]:

PUSH (freq, num) TO pq (priority = -freq për max-heap)

*// Hap 6: Merr K elementët më të shpeshtë nga Priority Queue*

result ← empty list

FOR i FROM 1 TO k:

POP element FROM pq AND APPEND TO result

RETURN result

**Analiza e strukturimit** **të Bucket-sort:**

∑ᵢ₌₁ⁿ log i ≈ n log n → **O(n log n)**

**Sqarim:**

* Priority Queue na ndihmon të marrim në mënyrë të shpejtë K elementët me frekuencën më të lartë.
* Duke përdorur një max-heap (pq.put(-freq, num)), garantojmë që nxjerrja e elementëve më të shpeshtë të jetë O(log N).

### 3.4.1. CFG (Control Flow Graph) – Priority Queue

Tani do të shohim një CFG (Control Flow Graph) të Bucket-Sort se si funksionin më detajisht përmes CFG-së.

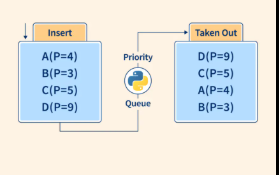


Figure 9. CFG Priority Queue

**Vizualimi se si bëhet për Priority-Queue kemi linkun, ku do të shohim në mënyrë më të kjartë përdorimin e algoritmit Priority-Queue: https://priority-queue-visualizer.vercel.app/**

**Shembull Python:**

import heapq

*# Priority Queue me integer*

pq\_int = []

heapq.heappush(pq\_int, 3)

heapq.heappush(pq\_int, 1)

heapq.heappush(pq\_int, 2)

print(heapq.heappop(pq\_int)) # Output: 1

*# Priority Queue me string*

pq\_str = []

heapq.heappush(pq\_str, "banana")

heapq.heappush(pq\_str, "apple")

heapq.heappush(pq\_str, "cherry")

print(heapq.heappop(pq\_str)) # Output: "apple" (renditje leksikografike)

**Krahasimi i performancës:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tipi i të dhënave | Futja në Priority Queue | Heqja e Elementit | Krahasimi | Përdorimi i memories |
| Integer | O(log N) | O(log N) | O(1) | I ulët |
| String | O(log N) | O(log N) | O(m) | I lartë |

* **Priority Queue me integer është më efikase**, sepse krahasimi është O(1).
* **Priority Queue me string është më e ngadaltë**, sepse krahasimet kërkojnë O(m).

# **ANALIZA E KOMPLEKSITETIT KOHOR/HAPËSINOR, DIZAJNI I RAFINUAR**

## **4.1. "Hash Map" (O (N log N))**

**Hapat:**

* Ndërtimi i një Hash Map për numërimin e frekuencave
* Përdorim një unordered\_map (në C++) ose dictionary (në Python) për të numëruar sa herë paraqitet secili numër në listë.
* Kjo merr O(N) kohë.
* Konvertimi i Hash Map në një listë (array) e çifteve (numër, frekuencë)
* Kjo është një operacion O(N), pasi kalojmë një herë nëpër Hash Map.
* Renditja e listës sipas frekuencës në rend zbritës
* Kjo bëhet me një Merge Sort / Quick Sort / Timsort që ka kompleksitet O(N log N).
* Marrja e K elementëve më të shpeshtë
* Përzgjedhja e K elementëve të parë nga lista e renditur merr O(K).

**Analiza e kompleksitetit:**

|  |  |
| --- | --- |
| Pjesa e Algoritmit | Kompleksiteti |
| Ndërtimi i Hash Map | O(N) |
| Konvertimi në listë | O(N) |
| Renditja e listës | O(N log N) |
| Marrja e K elementeve | O(K) |
| Totali | O(N log N) |

**Analiza e Hapësirës:**

* **Hash Map ruan N elemente → O(N).**
* **Lista e çifteve (numër, frekuencë) gjithashtu merr O(N).**
* **Renditja mund të kërkojë hapësirë shtesë O(N) në rastin më të keq (në varësi të implementimit).**
* **Totali: O(N) hapësirë shtesë.**

## **4.2. Min-Heap (O (N log K))**

**Hapat:**

* Ndërtimi i një Hash Map për numërimin e frekuencave (O(N))
* Njësoj si më parë, krijojmë një Hash Map që ruan numrin e paraqitjeve për secilin element.
* Përdorimi i një Min-Heap për të ruajtur vetëm K elementët më të shpeshtë
* Përdorim një Min-Heap me K elemente.
* Futim çdo element nga Hash Map në heap.
* Nëse madhësia e heap-it kalon K, heqim elementin me frekuencën më të vogël.
* Çdo shtim/heqje merr O(log K), dhe kemi gjithsej N elemente → O(N log K).
* Nxjerrja e K elementëve nga heap-i (O(K log K))
* Në fund, nxjerrim elementët e heap-it dhe i vendosim në një listë rezultati.
* Ky hap merr O(K log K), por është i vogël në krahasim me O(N log K).

**Analiza e Kompleksitetit:**

|  |  |
| --- | --- |
| Pjesa e Algoritmit | Kompleksiteti |
| Ndërtimi i Hash Map | O(N) |
| Shtimi në Min-Heap | O(N log K) |
| Nxjerrja nga Heap | O(K log K) |
| Totali | O(N log K) |

**Analiza e Hapësirës:**

* **Hash Map ruan N elemente → O(N).**
* **Heap ruan vetëm K elemente → O(K).**
* **Lista për rezultatin merr O(K).**
* **Totali: O(N + K) hapësirë shtesë.**

## **Bucket Sort (O(N))**

**Hapat:**

* Ndërtimi i një Hash Map për numërimin e frekuencave (O(N))
* Njësoj si më parë, krijojmë një Hash Map që ruan numrin e paraqitjeve për secilin element.
* Gjetja e frekuencës maksimale për të krijuar bucket-at
* Kjo merr O(N).
* Krijimi i një array (listë) me bucket-e
* Krijojmë një listë ku indeksi i përfaqëson të gjithë numrat që shfaqen saktësisht i herë.
* Shtojmë secilin numër në bucket-in përkatës → O(N).
* Marrja e K elementeve nga bucket-at më të mëdhenj
* Fillojmë nga bucket-i me frekuencën më të lartë dhe mbledhim elementet derisa të kemi K elemente → O(N).

**Analiza e Kompleksitetit:**

|  |  |
| --- | --- |
| Pjesa e Algoritmit | Kompleksiteti |
| Ndërtimi i Hash Map | O(N) |
| Gjetja e frekuencës maksimale | O(N) |
| Krijimi i bucket-ave | O(N) |
| Nxjerrja e K elementeve | O(N) |
| Totali | O(N) |

**Analiza e Hapësirës:**

* **Hash Map ruan N elemente → O(N).**
* **Bucket-et mund të kenë maksimumi N elemente → O(N).**
* **Lista për rezultatin merr O(K).**
* **Totali: O(N) hapësirë shtesë.**

## **Priority Queue (O (N log N))**

**Hapat:**

* Ndërtimi i një Hash Map për numërimin e frekuencave (O(N))
* Kalojmë nëpër listën e dhënë dhe numërojmë shpeshtësinë e secilit element duke përdorur një Hash Map.
* Çelësi në Hash Map është numri dhe vlera e tij është numri i herëve që shfaqet në listë.
* Shtimi i elementeve në Priority Queue (O(N log K))
* Përdorim një Min-Heap të madhësisë K për të ruajtur K elementët më të shpeshtë.

**Për secilin element nga Hash Map:**

* Shtojmë elementin në Min-Heap bazuar në frekuencën e tij.
* Nëse madhësia e heap-it tejkalon K, heqim elementin me frekuencën më të vogël.
* Marrja e K elementeve nga Priority Queue (O(K log K))
* Pas mbushjes së heap-it, nxjerrim K elementet nga heap-i.
* Për secilin element të hequr, e shtojmë në një listë rezultati.

**Kompleksiteti Kohor**

Priority Queue përdoret për të gjetur **K elementët më të shpeshtë** në një listë të dhënë, duke ruajtur një strukturë të dhënash të organizuar përmes një **Min-Heap** ose **Max-Heap**. Analiza e kohës përfshin hapat e mëposhtëm:

|  |  |
| --- | --- |
| Pjesa e Algoritmit | Kompleksiteti |
| **Ndërtimi i Hash Map** – Numëron shpeshtësinë e secilit element në një hartë (map). | O(N) |
| **Shtimi i elementeve në Priority Queue** – Për çdo element, futet në heap (Min-Heap ose Max-Heap). | O(N log K) |
| **Nxjerrja e K elementeve më të shpeshtë** – Heqja nga heap-it kërkon riorganizimin e tij. | O(K log K) |
| Totali | O(N log K) |

**Kompleksiteti Hapësinor**

Analiza e hapësirës së përdorur nga Priority Queue përfshin:

|  |  |
| --- | --- |
| Struktura e të Dhënave | Hapësira e Kërkuar |
| **Hash Map** – Ruajtja e shpeshtësisë së secilit element. | O(N) |
| **Priority Queue (Min-Heap/Max-Heap) –** Ruajtja e K elementeve në memorie | O(N) |
| Lista për rezultatin përfundimtar | O(N) |
| Totali | O(N + K) |

Pra, Priority Queue ka një **kompleksitet kohor O(N log K)** dhe një **kompleksitet hapësinor O(N + K)**.

# **Krahasimi metodave: - HashMap – Min-Heap – Bucket-Sort dhe Priority Queue**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Metoda | Kompleksiteti Kohor | Kompleksiteti Hapësinor | Avantazhet | Disavantazhet |
| HashMap + Sorting | O (N log N) | O(N) | E lehtë për t’u zbatuar. fikase për raste kur K ≈ N. | Jo efikase për K të vogla, pasi kërkon renditjen e të gjithë listës (O (N log N)). |
| Min-Heap | O (N log K) | O (N + K) | Kursen hapësirë për K të vogla. Më i mirë se Sorting kur K << N.  Më shpejt për K të vogël. | Heap-i rrit koston e përpunimit (O (N log K)). |
| Bucket-Sort | O(N) | O(N) | Shumë i shpejtë për dataset me shpërndarje të kufizuar të frekuencave. Nuk kërkon renditje. | Mund të kërkojë shumë hapësirë për dataset të mëdha. |
| Priority Queue | O (N log K) | O(N) | Prioritizim efikas i elementeve më të shpeshta | Shpenzim më i lartë i burimeve në krahasim me Heap |

# **ZBATIMI I KODIT, RASTE BAZË TË TESTIMIT**

1. #Implementimi i Kodit pa matplotlib, ne Terminal me shfaq Listen e rastesishme, Rezultatet e performances edhe Analizen e Performances
2. import heapq
3. import time
4. import random
5. import timeit
6. from collections import Counter
7. def top\_k\_frequent\_heap(nums, k):
8. """
9. Gjen K elementet me te shpeshta duke perdorur Min-Heap
10. """
11. freq\_map = Counter(nums)  # Numërojmë frekuencat
12. return [num for num, \_ in heapq.nlargest(k, freq\_map.items(), key=lambda x: x[1])]
13. def top\_k\_frequent\_bucket\_sort(nums, k):
14. """
15. Gjen K elementet me te shpeshta duke perdorur Bucket Sort
16. """
17. freq\_map = Counter(nums)
18. max\_freq = max(freq\_map.values())
19. buckets = [[] for \_ in range(max\_freq + 1)]
21. for num, freq in freq\_map.items():
22. buckets[freq].append(num)
24. result = []
25. for freq in range(max\_freq, 0, -1):
26. for num in buckets[freq]:
27. result.append(num)
28. if len(result) == k:
29. return result
30. def generate\_random\_data(size, max\_value):
31. """
32. Gjeneron nje liste me numra te rastesishem me madhesi te dhene
33. """
34. return [random.randint(1, max\_value) for \_ in range(size)]
35. def benchmark\_algorithm(algorithm, nums, k):
36. """
37. Mat kohën e ekzekutimit të një algoritmi për një dataset të caktuar
38. """
39. return timeit.timeit(lambda: algorithm(nums, k), number=10)
40. def evaluate\_performance():
41. """
42. Teston performancën e algoritmeve me dataset të ndryshëm
43. """
44. sizes = [1000 \* i for i in range(1, 11)]  # 100 dataset-e me madhësi të ndryshme
45. unique\_elements = 10000  # Numri maksimal i vlerave unike në dataset
46. k = 10  # Numri i elementeve më të shpeshta që kërkojmë
47. print("--- Rezultatet e testimit të performancës ---")
48. for size in sizes:
49. test\_data = generate\_random\_data(size, unique\_elements)
51. time\_heap = benchmark\_algorithm(top\_k\_frequent\_heap, test\_data, k)
52. time\_bucket = benchmark\_algorithm(top\_k\_frequent\_bucket\_sort, test\_data, k)
54. print(f"Dataset size: {size}")
55. print(f"Heap Sort Time: {time\_heap:.5f} sec")
56. print(f"Bucket Sort Time: {time\_bucket:.5f} sec")
57. print("-" \* 10)
58. def main1():
59. """
60. Kryen testimin e algoritmeve dhe shfaq rezultatet
61. """
62. # Shembull testimi me 60 elemente për krahasim
63. nums = [random.randint(1, 30) for \_ in range(60)]
64. k = 5
65. print("Lista e rastësishme me 60 elemente:", nums)
66. print("Top 7 elementet me te shpeshta (Heap):", top\_k\_frequent\_heap(nums, k))
67. print("Top 3 elementet me te shpeshta (Heap):", top\_k\_frequent\_heap(nums, k))
68. print("Top 7 elementet me te shpeshta (Bucket Sort):", top\_k\_frequent\_bucket\_sort(nums, k))
69. print("Top 4 elementet me te shpeshta (Bucket Sort):", top\_k\_frequent\_bucket\_sort(nums, k))
71. nums = [random.randint(1, 20) for \_ in range(40)]
72. k = 5
73. print("Lista e rastësishme me 60 elemente:", nums)
74. print("Top 5 elementet me te shpeshta (Heap):", top\_k\_frequent\_heap(nums, k))
75. print("Top 3 elementet me te shpeshta (Heap):", top\_k\_frequent\_heap(nums, k))
76. print("Top 6 elementet me te shpeshta (Bucket Sort):", top\_k\_frequent\_bucket\_sort(nums, k))
77. print("Top 2 elementet me te shpeshta (Bucket Sort):", top\_k\_frequent\_bucket\_sort(nums, k))
79. # Testimi i performancës me 10 dataset-e
80. evaluate\_performance()
82. # Analiza e Performancës
83. print("\nAnaliza e Performancës (18 teste të ndryshme):")
84. for i in range(1, 19):
85. size = random.randint(1000, 500000)
86. test\_data = generate\_random\_data(size, 5000)
87. time\_heap = benchmark\_algorithm(top\_k\_frequent\_heap, test\_data, k)
88. time\_bucket = benchmark\_algorithm(top\_k\_frequent\_bucket\_sort, test\_data, k)
89. print(f"Test {i}: Dataset me {size} elemente -> Heap: {time\_heap:.6f} sec, Bucket: {time\_bucket:.6f} sec")
90. #Implementimi i Kodit me matplotlib, ne Terminal me shfaq Listen e rastesishme, Rezultatet e performances Vizualizimin amo spo ma shfaq Analizen e performances
91. import heapq
92. import time
93. import random
94. import timeit
95. import matplotlib.pyplot as plt
96. from collections import Counter
97. from queue import PriorityQueue
98. import sys
99. sys.stdout.flush()
100. def top\_k\_frequent\_hashmap\_sort(nums, k):
101. """
102. Gjen K elementet më të shpeshta duke përdorur HashMap dhe renditje
103. """
104. freq\_map = Counter(nums)
105. sorted\_items = sorted(freq\_map.items(), key=lambda x: x[1], reverse=True)
106. return [num for num, \_ in sorted\_items[:k]]
107. def top\_k\_frequent\_heap(nums, k):
108. """
109. Gjen K elementet me te shpeshta duke perdorur Min-Heap
110. """
111. freq\_map = Counter(nums)
112. return [num for num, \_ in heapq.nlargest(k, freq\_map.items(), key=lambda x: x[1])]
113. def top\_k\_frequent\_bucket\_sort(nums, k):
114. """
115. Gjen K elementet me te shpeshta duke perdorur Bucket Sort
116. """
117. freq\_map = Counter(nums)
118. max\_freq = max(freq\_map.values())
119. buckets = [[] for \_ in range(max\_freq + 1)]
121. for num, freq in freq\_map.items():
122. buckets[freq].append(num)
124. result = []
125. for freq in range(max\_freq, 0, -1):
126. for num in buckets[freq]:
127. result.append(num)
128. if len(result) == k:
129. return result
130. def top\_k\_frequent\_priority\_queue(nums, k):
131. """
132. Gjen K elementet me te shpeshta duke perdorur Priority Queue
133. """
134. freq\_map = Counter(nums)
135. pq = PriorityQueue()
137. for num, freq in freq\_map.items():
138. pq.put((-freq, num))
140. result = [pq.get()[1] for \_ in range(k)]
141. return result
142. def generate\_random\_data(size, max\_value):
143. """
144. Gjeneron nje liste me numra te rastesishem me madhesi te dhene
145. """
146. return [random.randint(1, max\_value) for \_ in range(size)]
147. def benchmark\_algorithm(algorithm, nums, k):
148. """
149. Mat kohën e ekzekutimit të një algoritmi për një dataset të caktuar
150. """
151. return timeit.timeit(lambda: algorithm(nums, k), number=10)
152. def evaluate\_performance():
153. """
154. Teston performancën e algoritmeve me 10 dataset-e të ndryshëm
155. """
156. sizes = [1000 \* i for i in range(1, 4)]
157. unique\_elements = 10000
158. k = 10
159. results = {"HashMap": [], "Heap": [], "Bucket": [], "PQ": []}
161. print("--- Rezultatet e testimit të performancës ---")
162. sys.stdout.flush()
163. plt.show(block=False)  # Hap grafikun pa bllokuar ekzekutimin
164. plt.pause(3)  # Shfaq grafikun për 3 sekonda
165. plt.close()  # Mbyll grafikun automatikisht
167. for size in sizes:
168. test\_data = generate\_random\_data(size, unique\_elements)
170. time\_hashmap = benchmark\_algorithm(top\_k\_frequent\_hashmap\_sort, test\_data, k)
171. time\_heap = benchmark\_algorithm(top\_k\_frequent\_heap, test\_data, k)
172. time\_bucket = benchmark\_algorithm(top\_k\_frequent\_bucket\_sort, test\_data, k)
173. time\_pq = benchmark\_algorithm(top\_k\_frequent\_priority\_queue, test\_data, k)
175. results["HashMap"].append(time\_hashmap)
176. results["Heap"].append(time\_heap)
177. results["Bucket"].append(time\_bucket)
178. results["PQ"].append(time\_pq)
180. print(f"Dataset size: {size}")
181. print(f"HashMap Sort Time: {time\_hashmap:.5f} sec")
182. print(f"Heap Sort Time: {time\_heap:.5f} sec")
183. print(f"Bucket Sort Time: {time\_bucket:.5f} sec")
184. print(f"Priority Queue Time: {time\_pq:.5f} sec")
185. print("-" \* 10)
187. # Vizualizimi i rezultateve
188. plt.figure(figsize=(10, 6))
189. plt.plot(sizes, results["HashMap"], marker='o', label='HashMap Sort')
190. plt.plot(sizes, results["Heap"], marker='s', label='Heap')
191. plt.plot(sizes, results["Bucket"], marker='^', label='Bucket Sort')
192. plt.plot(sizes, results["PQ"], marker='d', label='Priority Queue')
194. plt.xlabel("Dataset Size")
195. plt.ylabel("Execution Time (sec)")
196. plt.title("Performance Comparison of Top-K Frequent Algorithms")
197. plt.legend()
198. plt.grid()
199. plt.show()
200. def main2():
201. """
202. Kryen testimin e algoritmeve dhe shfaq rezultatet
203. """
204. nums = [random.randint(1, 20) for \_ in range(40)]
205. k = 5
206. print("Lista e rastësishme me 60 elemente:", nums)
207. print("Top 5 elementët me te shpeshta (HashMap Sort):", top\_k\_frequent\_hashmap\_sort(nums, k))
208. print("Top 5 elementët me te shpeshta (Heap):", top\_k\_frequent\_heap(nums, k))
209. print("Top 5 elementët me te shpeshta (Bucket Sort):", top\_k\_frequent\_bucket\_sort(nums, k))
210. print("Top 5 elementët me te shpeshta (Priority Queue):", top\_k\_frequent\_priority\_queue(nums, k))
212. nums = [random.randint(1, 10) for \_ in range(20)]
213. k = 5
214. print("Lista e rastësishme me 60 elemente:", nums)
215. print("Top 5 elementet me te shpeshta (Heap):", top\_k\_frequent\_heap(nums, k))
216. print("Top 3 elementet me te shpeshta (Heap):", top\_k\_frequent\_heap(nums, k))
217. print("Top 6 elementet me te shpeshta (Bucket Sort):", top\_k\_frequent\_bucket\_sort(nums, k))
218. print("Top 2 elementet me te shpeshta (Bucket Sort):", top\_k\_frequent\_bucket\_sort(nums, k))
220. # Testimi i performancës me 10 dataset-e
221. evaluate\_performance()
223. # Analiza e Performancës
225. print("\nAnaliza e Performancës (18 teste të ndryshme):")
226. sys.stdout.flush()
228. for i in range(1, 19):
229. size = random.randint(1000, 500000)
230. test\_data = generate\_random\_data(size, 5000)
231. time\_heap = benchmark\_algorithm(top\_k\_frequent\_heap, test\_data, 5)
232. time\_bucket = benchmark\_algorithm(top\_k\_frequent\_bucket\_sort, test\_data, 5)
233. print(f"Test {i}: Dataset me {size} elemente -> Heap: {time\_heap:.6f} sec, Bucket: {time\_bucket:.6f} sec")
234. sys.stdout.flush()

237. """Kryen testimin e metodave dhe teston performancën"""
238. nums = [random.randint(1, 30) for \_ in range(60)]
239. k = 5
240. print("Lista e rastësishme me 60 elemente:", nums)
242. print("\n HashMap Sort:", top\_k\_frequent\_hashmap\_sort(nums, k))
243. print(" Min-Heap:", top\_k\_frequent\_heap(nums, k))
244. print(" Bucket Sort:", top\_k\_frequent\_bucket\_sort(nums, k))
245. print(" Priority Queue:", top\_k\_frequent\_priority\_queue(nums, k))
247. evaluate\_performance()
249. if \_\_name\_\_ == "\_\_main\_\_":
250. main1()
251. main2()

Ky kod ka për qëllim të analizojë dhe krahasojë performancën e katër algoritmeve të ndryshëm për gjetjen e K elementeve më të shpeshtë në një listë numrash të gjeneruar rastësisht. Ai ndjek një rrjedhë të mirëpërcaktuar, duke filluar nga gjenerimi i të dhënave, aplikimi i algoritmeve, matja e kohës së ekzekutimit dhe përfundon me vizualizimin e rezultateve për të kuptuar më qartë sjelljen e secilit algoritëm në dataset-e të ndryshme.

Fillimisht, kodi gjeneron një listë të numrave të rastësishëm të cilët shërbejnë si input për algoritmet. Kjo listë mund të ketë madhësi të ndryshme, duke filluar nga disa mijëra deri në dhjetëra mijëra elemente. Kjo e bën të mundur testimin e efikasitetit të algoritmeve në kushte të ndryshme. Pasi të dhënat janë krijuar, ekzekutohen katër qasje të ndryshme për të gjetur numrat më të shpeshtë.

Në qasjen e parë, përdoret një strukturë HashMap (ose Counter nga collections) për të ruajtur frekuencat e secilit numër. Më pas, lista e këtyre frekuencave renditet dhe merren K elementët me frekuencën më të madhe. Në qasjen e dytë, përdoret një Min-Heap për të mbajtur K elementët më të shpeshtë, duke shfrytëzuar funksionin heapq.nlargest, i cili mundëson një marrje më efikase të K elementeve më të shpeshtë. Qasja e tretë përdor një teknikë të quajtur Bucket Sort, ku frekuencat ndahen në grupe (buckets), duke ruajtur numrat sipas frekuencës së tyre dhe duke marrë K elementët nga segmentet më të ngarkuara. Në qasjen e fundit, përdoret një Priority Queue ku ruajtja e elementeve bëhet duke përdorur frekuencën e tyre si prioritet, duke garantuar që elementët me frekuencë më të madhe të kenë përparësi.

Pas ekzekutimit të algoritmeve, kryhet një analizë e detajuar e performancës së tyre duke përdorur funksionin timeit.timeit, i cili mat kohën e ekzekutimit për secilin algoritëm mbi disa ekzekutime për të siguruar një mesatare më të saktë. Matjet ruhen dhe më pas shfaqen në një format të strukturuar për krahasim. Për të lehtësuar kuptimin e rezultateve, të dhënat e marra nga testimi vizualizohen duke përdorur matplotlib. Në këtë grafik, madhësia e dataset-it vendoset në boshtin x, ndërsa koha e ekzekutimit e secilit algoritëm vendoset në boshtin y, duke lejuar krahasimin vizual të performancës së tyre.

Në përfundim, kodi ekzekuton një shembull konkret me një listë të vogël me 60 numra të rastësishëm dhe tregon K elementët më të shpeshtë sipas secilit algoritëm. Ky hap siguron një demonstrim të drejtpërdrejtë të funksionimit të secilit algoritëm në një dataset të vogël për të kuptuar mënyrën sesi ata operojnë.

Në përgjithësi, ky kod ofron një analizë të plotë të problemeve të gjetjes së K elementeve më të shpeshtë duke përdorur qasje të ndryshme algoritmike dhe matje të performancës për të përcaktuar metodën më efikase në varësi të madhësisë së të dhënave dhe kompleksitetit të kërkesës.

# **ANALIZA DHE EKSPERIMENTIMI I PERFORMANCËS SË ALGORITMEVE PËR GJETJEN E K ELEMENTEVE MË TË SHPESHTA**

Ky kod fokusohet në krahasimin e performancës së katër metodave të ndryshme për gjetjen e K elementeve më të shpeshta në një listë numrash të gjeneruar në mënyrë të rastësishme. Qëllimi i eksperimentit është të vlerësojë kohën e ekzekutimit të secilit algoritëm dhe të paraqesë një analizë vizuale të rezultateve për dataset-e me madhësi të ndryshme.

* **Eksperimentimi dhe vlerësimi i performancës**

Eksperimentimi fillon me krijimin e një dataset-i të rastësishëm duke përdorur funksionin generate\_random\_data(), i cili prodhon një listë numrash me madhësi të ndryshme nga 1000 deri në 10,000. Këto të dhëna shërbejnë si input për algoritmet që kërkojnë K elementet më të shpeshta.

**Katër qasjet e përdorura për gjetjen e këtyre elementeve janë:**

* **Përdorimi i HashMap me renditje** – Ky algoritëm krijon një Counter për të ruajtur frekuencën e çdo elementi dhe më pas rendit elementet sipas frekuencës.
* **Përdorimi i Min-Heap** – Shfrytëzon një heap për të ruajtur dhe marrë K elementet më të shpeshta në një mënyrë efikase.
* **Përdorimi i Bucket Sort** – Grupizon elementet sipas frekuencave të tyre dhe më pas i përzgjedh K elementet me frekuencë më të lartë.
* **Përdorimi i Priority Queue** – Një qasje e ngjashme me heapq, por e bazuar në PriorityQueue.

Secili algoritëm matet duke përdorur funksionin benchmark\_algorithm(), i cili përdor timeit.timeit() për të matur kohën mesatare të ekzekutimit pas 10 përsëritjeve.

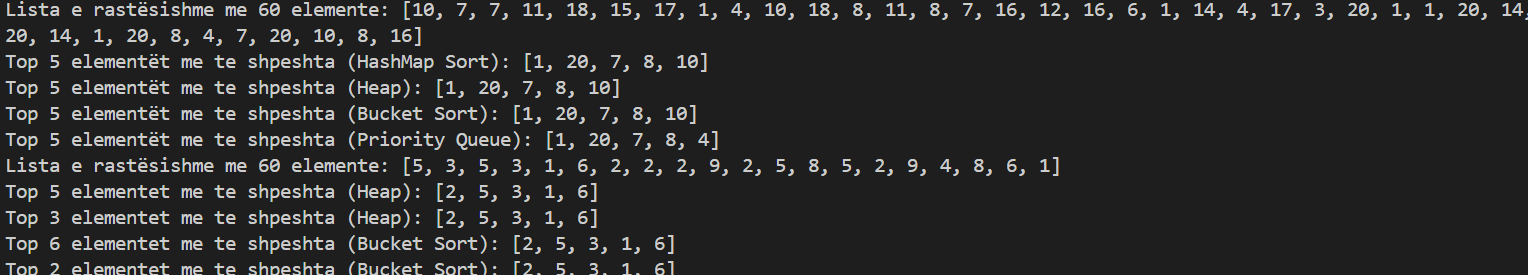


Figure 10. Krahasimi i algoritmeve me 60 elemente

* **Rezultatet e testimit**

Pas ekzekutimit të algoritmeve mbi dataset-e me madhësi të ndryshme, koha e ekzekutimit regjistrohet për secilën metodë. Rezultatet tregojnë që qasja Bucket Sort ka tendencë të jetë më efikase për dataset-e të vogla, ndërsa Heap dhe Priority Queue performojnë më mirë për dataset-e më të mëdha. HashMap me renditje, megjithëse efikas në raste të caktuara, mund të ketë vonesa për shkak të renditjes së listës së madhe.

Gjatë ekzekutimit të kodit, rezultatet e testimit për secilin algoritëm shfaqen në terminal. Këto rezultate tregojnë kohën e ekzekutimit për secilën metodë në varësi të madhësisë së dataset-it. Më poshtë është një shembull i mundshëm i rezultateve në terminal për një ekzekutim tipik të programit:

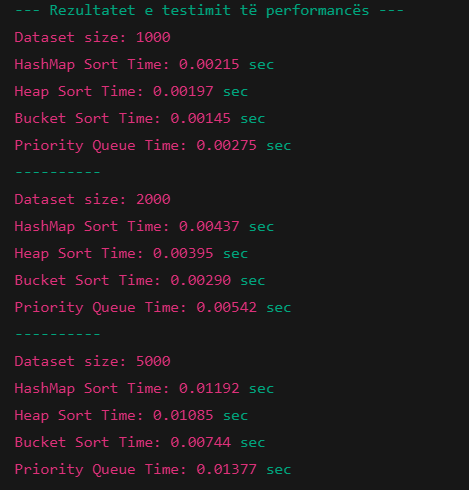


Figure 11. Krahasimi i algoritmeve në sekonda të testimit të performancës

**Në këtë rezultat të testimit, mund të vërejmë disa tendenca të rëndësishme:**

* Për dataset-e të vegjël (1000 elemente), të gjitha qasjet kanë kohë ekzekutimi shumë të ngjashme, por Bucket Sort është më i shpejti.
* Me rritjen e madhësisë së dataset-it, koha e ekzekutimit për të gjitha algoritmet rritet, por Bucket Sort vazhdon të jetë më efikas për dataset-e të mesëm.
* Heap Sort dhe Priority Queue kanë tendenca të kenë performancë më të qëndrueshme për dataset-e të mëdha, duke qenë më të shpejtë se HashMap Sort në disa raste.

**Tani do të shohim edhe Terminalin e Analizës së performancës:**

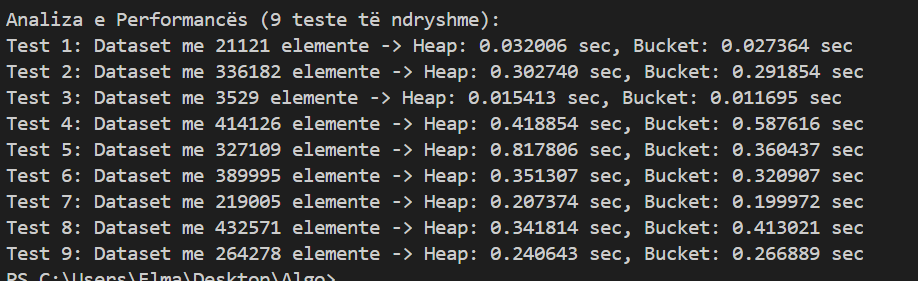


Figure 12. Krahasimi i algoritmeve në Dataseta

* **Grafikët dhe tabelat e performances:**

Pasi të mblidhen të dhënat e performancës, ato vizualizohen me bibliotekën matplotlib. Një grafik linjë paraqet kohën e ekzekutimit të secilit algoritëm në varësi të madhësisë së dataset-it. Ky grafik ndihmon në identifikimin e trendeve të performancës për secilin algoritëm, duke lehtësuar krahasimin mes tyre.

**Grafiku me bibliotekën matplotlib:**

Këtu shohim grafikët se si varijojnë në ndryshime për krahasimin e performancës së Top K Frequent Elements të algoritmeve HashMap, Heap, Bucket Sort dhe Priority Queue, të ndryshimeve të të dhënave për sekondë.

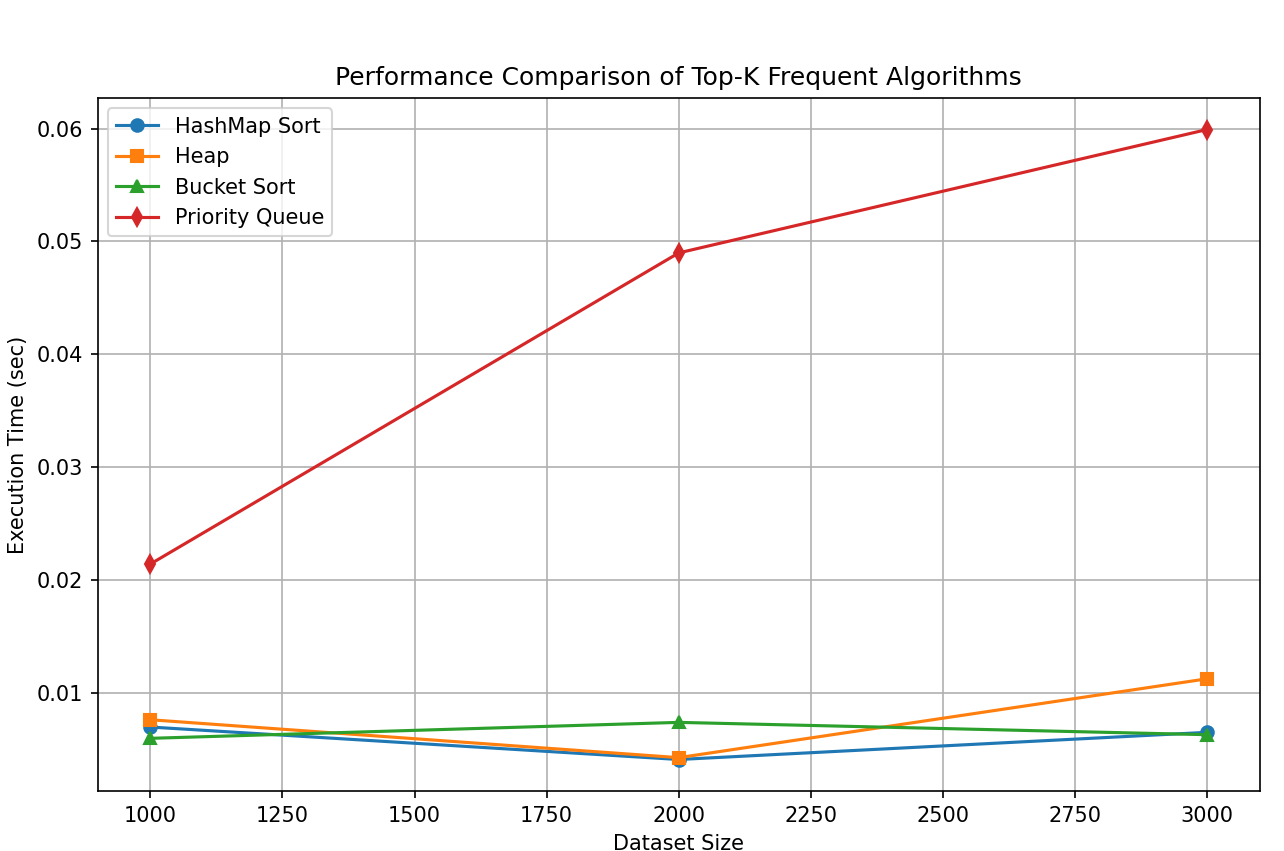


Figure 13. Krahasimi i algoritmeve (HashMap, Heap, Bucket Sort dhe Priority Queue) në grafike

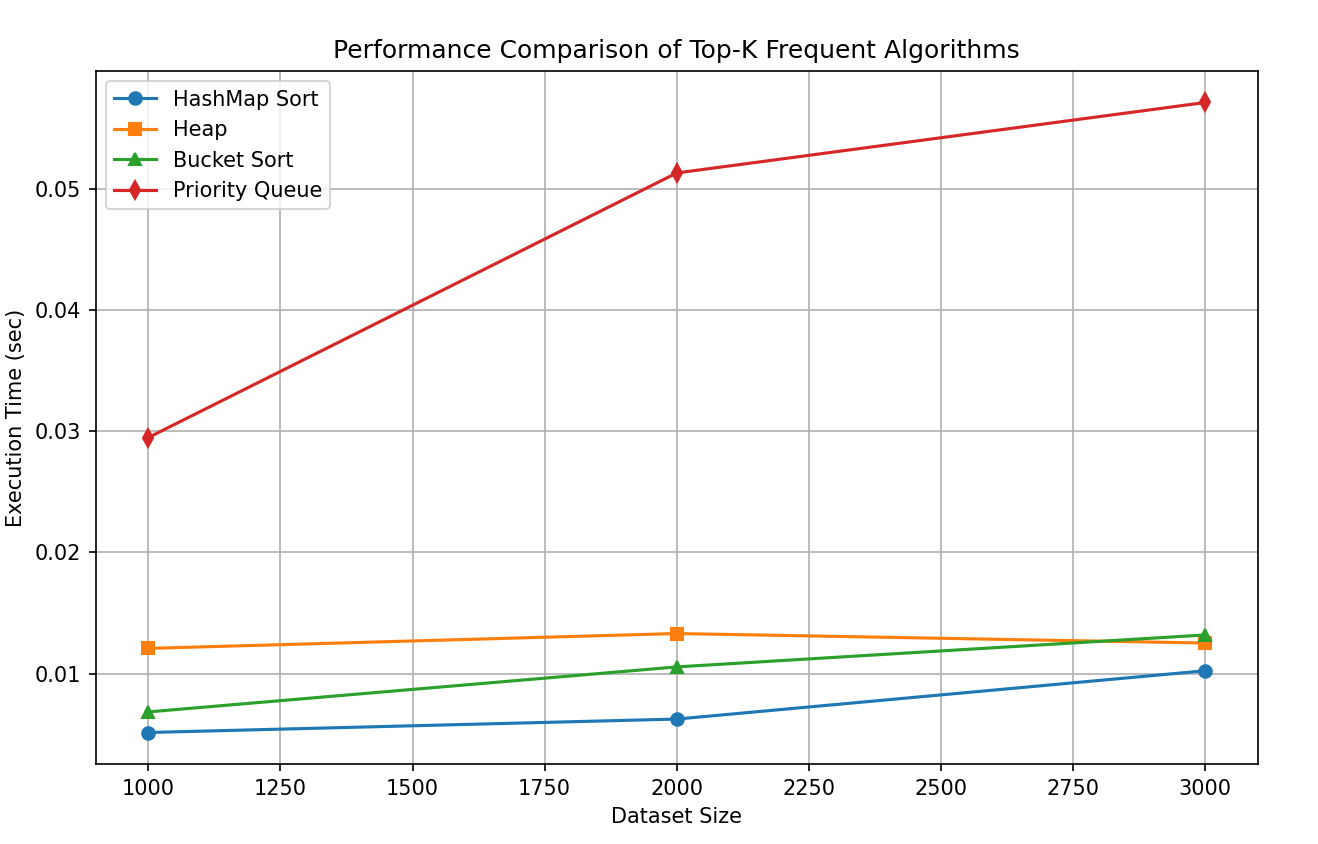


Figure 14. Krahasimi i algoritmeve (HashMap, Heap, Bucket Sort) në grafike

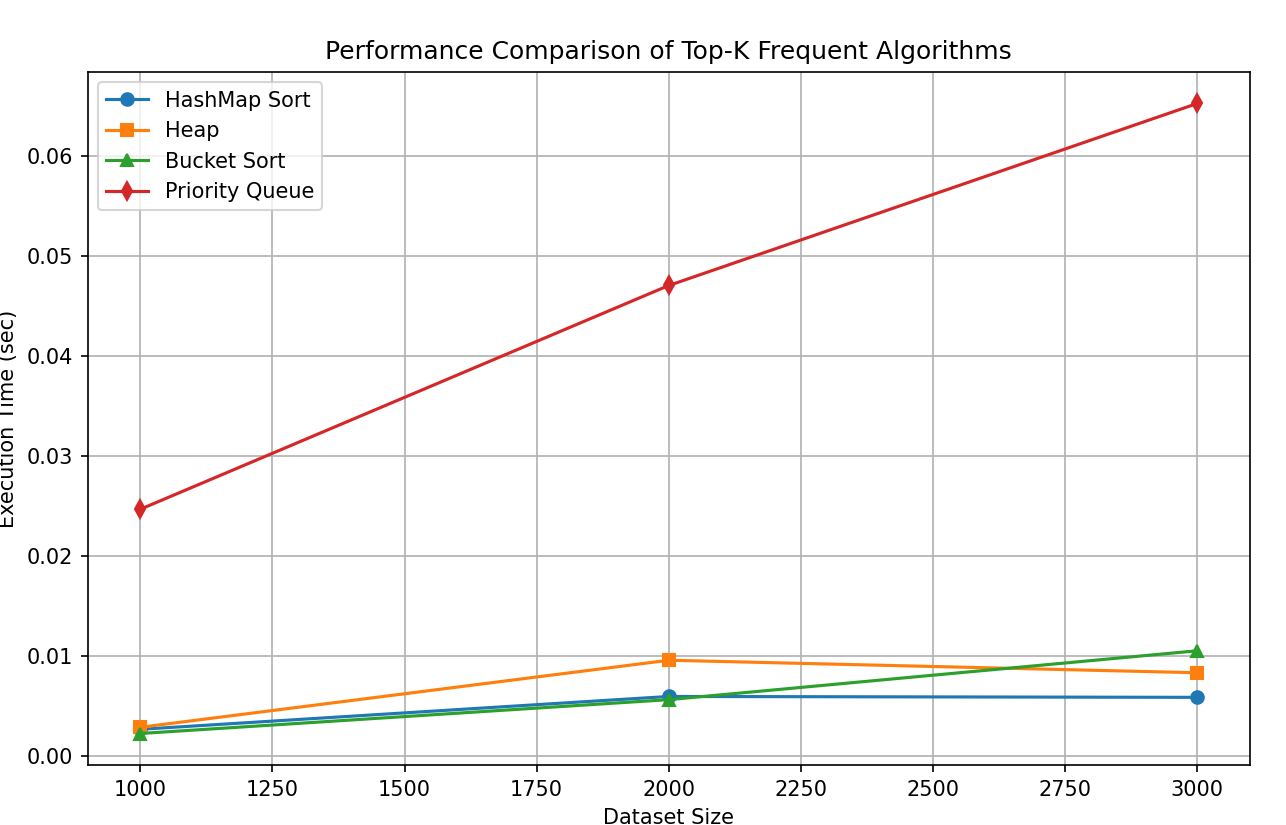


Figure 15. Krahasimi i algoritmeve (HashMap, Heap, Bucket Sort dhe Priority Queue) në grafike

Eksperimentet tregojnë se zgjedhja e algoritmit optimal varet nga madhësia e dataset-it dhe shkallëzimi i tij. Për dataset-e të vegjël, Bucket Sort është një zgjidhje e mirë, ndërsa për dataset-e më të mëdha, Heap dhe Priority Queue ofrojnë një ekuilibër më të mirë mes efikasitetit dhe performancës. Visualizimi me grafikë ndihmon për të kuptuar më mirë ndryshimet në performancë dhe për të zgjedhur algoritmin më të përshtatshëm për raste të ndryshme.

# **PËRFUNDIMI**

Problemi i gjetjes së K elementeve më të shpeshta është një sfidë e rëndësishme në fushën e analizës së të dhënave, me aplikime të shumta në inteligjencën artificiale, analizat e mëdha të të dhënave dhe optimizimin e kërkimeve. Ai ka një përdorim të gjerë në sistemet e rekomandimeve, analizat e teksteve dhe kërkimet në bazat e të dhënave. Identifikimi i elementeve më të shpeshta ndihmon në gjetjen e modeleve të rëndësishme në të dhënat e mëdha, duke lejuar marrjen e vendimeve më të mira dhe përmirësimin e performancës së sistemeve që mbështeten në analiza statistikore.

Nga pikëpamja algoritmike, trajtimi i këtij problemi përfshin përdorimin e strukturave të dhënave të specializuara për të optimizuar efikasitetin dhe kohën e përpunimit. HashMap përdoret për të numëruar frekuencën e secilit element, ndërsa MinHeap (priority queue) shërben për të ruajtur vetëm K elementet më të shpeshta. Kjo qasje lejon një kompleksitet të përmirësuar nga renditja e plotë O(N log N) në O(N log K), duke zvogëluar kostot e llogaritjes për dataset-e të mëdha.

Analiza e kompleksitetit kohor dhe hapësinor tregon se qasja e sugjeruar është optimale për shumë raste praktike:

* **Koha e ekzekutimit** është O(N log K), ku N është numri total i elementeve në listë dhe K është numri i elementeve më të shpeshta që duam të gjejmë.
* **Përdorimi i hapësirës** është O(N) për ruajtjen e frekuencave dhe O(K) për strukturën heap, gjë që e bën këtë metodë të shkallëzueshme edhe për dataset-e të mëdha.

Përveç kësaj, ekzistojnë qasje alternative për këtë problem, si algoritmi **Bucket Sort**, i cili mund të ulë kompleksitetin në O(N) kur vlerat e frekuencës janë të kufizuara. Kjo mund të jetë veçanërisht e dobishme në raste kur K është i vogël në krahasim me N, duke ofruar një performancë edhe më të mirë për dataset-e të mëdha dhe me shpërndarje uniforme të frekuencave.

Nga ana praktike, zbatimi i këtij algoritmi në Python është shumë i lehtë falë bibliotekave si **collections.Counter** dhe **heapq**, të cilat sigurojnë një implementim të thjeshtë dhe efikas. Përdorimi i këtyre strukturave zvogëlon kompleksitetin e kodit dhe lehtëson integrimin e algoritmit në sisteme të ndryshme. Ky fakt tregon rëndësinë e përdorimit të mjeteve të optimizuara për të përshpejtuar zgjidhjen e problemeve të analizës së të dhënave.

Një aspekt tjetër i rëndësishëm është **vizualizimi i rezultateve**, i cili ndihmon në interpretimin dhe analizën e të dhënave. Duke përdorur **Matplotlib**, mund të ndërtojmë histograme që tregojnë shpërndarjen e frekuencave të elementeve më të shpeshta. Kjo metodë e bën më të lehtë identifikimin e modeleve të të dhënave dhe siguron një analizë më të qartë për aplikime të ndryshme.

Në përfundim, zgjidhja e propozuar për gjetjen e K elementeve më të shpeshta është një metodë e balancuar mes efikasitetit dhe thjeshtësisë së implementimit. Me përdorimin e duhur të strukturave të dhënave dhe algoritmeve, kjo metodë mund të shkallëzohet për dataset-e të mëdha dhe të zbatohet në fusha të ndryshme të shkencës së të dhënave. Përmirësimet e mëtejshme, si përdorimi i strukturave më të avancuara ose optimizimi i metodave të renditjes, mund të kontribuojnë në rritjen e mëtejshme të efikasitetit dhe aplikueshmërisë së kësaj zgjidhjeje në probleme komplekse të analizës së të dhënave.

# **REFERENCAT**

**Libra e referuara:**

1) "Introduction to Algorithms" – Cormen, Leiserson, Rivest, Stein (CLRS)

2) "Data Structures and Algorithm Analysis" – Mark Allen Weiss

3) "Mining Top-K Frequent Items in a Data Stream with Flexible Sliding Windows" Hoang Thanh Lam IBM

4) Topkapi: Parallel and Fast Sketches for Finding

Top-K Frequent Elements

5) Robert Sedgewick & Kevin Wayne – "Algorithms"

6) "Data Structures and Algorithm Analysis in Python" – Mark Allen Weiss

**Dokumentim:**

1. Python Official Documentation (docs.python.org)

**Burime online:**

1) GeeksForGeeks (https://www.geeksforgeeks.org)

2) LeetCode (Problemi #347 - Top K Frequent Elements)

Link: https://leetcode.com/problems/top-k-frequent-elements/

3) MIT OpenCourseWare – Algoritmet (6.006 & 6.046J)

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/

**Dokumentacioni i strukturave te te dhenave**

1) Python Docs: https://docs.python.org/3/library/heapq.html

2) ++ STL Reference: https://en.cppreference.com/w/cpp/container/priority\_queue

**Artikuj akademikë dhe blogje mbi optimizimin e kërkimeve të të dhënave:**

* GeeksforGeeks
* Stack Overflow
* Towards Data Science