**UNIVERSITÀ DEGLI STUDI DI UDINE** DIPARTIMENTO DI SCIENZE MATEMATICHE, INFORMATICHE E FISICHE

**Docenti:**

Chiar.mo Gabriele Puppis

Chiar.ma Carla Piazza

**ANNO ACCADEMICO 2023/2024**

**QuickSelect**

Lo scopo è quello di implementare una variante dell’algoritmo di ordinamento di QuickSort, in cui ogni chiamata ricorsiva su un intervallo del vettore utilizzato in input, termina restituendo l’ i-esimo elemento più piccolo del vettore qualora , oppure un valore indefinito qualora dove, nel secondo caso, una chiamata sull’intervallo può terminare senza l’utilizzo di Partition.

Inoltre, l’algoritmo dovrà avere una complessità di nel caso peggiore, mentre nel caso medio, dove n indicherà il numero di elementi appartenente al vettore

**PseudoCodice**

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

**Complessità di QuickSelect**

|  |  |  |
| --- | --- | --- |
| Analisi del codice | **Costo nel caso medio** | **Costo nel caso peggiore** |
| **if**(*k****<****0* **or** *k* ***>*** **len**(*arr*) ***-*** *1*): |  |  |
| **if**(l**==**r): |  |  |
| p\_index = **partition**(arr, l, r) |  |  |
| **if** (k **==** p\_index): **return** arr[p\_index] |  |  |
| **elif** (k **<** p\_index): **return** **QuickSelect**(arr, l, p\_index - 1, k) |  |  |
| **else**: **return** **QuickSelect**(arr, p\_index + 1, r, k) |

**Analisi dei casi:**

La complessita temporale di QuickSelect è strettamente influenzata dalla scelta del **pivot** e quindi dalla peculiarità del partizionamento. Questo perché a seconda della sua posizione, i sotto-array variano di dimensione.

* **Caso medio:** il pivot è scelto in modo randomico e in media (complessivamente sui test) l’array viene diviso in due parti di dimensioni più o meno simili. L’equazione ricorsiva di complessità risulta essere: l’ultimo termine dell’equazione è il costo di Partition

Risolvendo l’equazione, si ricava il costo complessivo, ossia .

* **Caso peggiore:** la scelta del pivot sbilancia il partizionamento dell’array. Il pivot è uno tra gli elementi minori o maggiori dell’array e gli elementi restanti non vengono partizionati uniformemente, ma restano in un sotto-array più grande o più piccolo. L’equazione ricorsiva in questo caso è:

Risolvendo l’equazione, ricaviamo un costo pari a .

**Correttezza di QuickSelect**

CASO BASE

Entriamo nel caso base di QuickSelect quando l’indice sinistro dell’intervallo è uguale all’indice destro , quindi quando contiene un solo elemento. Se quindi, restituiamo l’elemento dell’array in posizione .

PASSO INDUTTIVO:

Ora passiamo all’analisi di nei 3 casi successivi al caso base, considerando che la lunghezza dell’array sia ora pari a

* Primo caso:

Dopo aver effettuato il partizionamento dell’array (o di un sotto-array), scopriamo che il pivot che abbiamo trovato precedentemente ha lo stesso indice salvato in . Restituiamo e concludiamo la procedura.

* Secondo caso:

Dopo il partizionamento, se il valore di e’ minore dell’indice del pivot, allora sappiamo che il k-esimo elemento piu’ piccolo si trova nell’intervallo che va da a – 1.

Ipotizziamo che richiamando ricorsivamente la funzione su quell’intervallo, partizioniamo ulteriormente l’array fino a trovare .

* Terzo caso:

Dopo il partizionamento, se il valore di e’ maggiore dell’indice del pivot, allora sappiamo che il k-esimo elemento piu’ piccolo si trova nell’intervallo che va da + 1 a .

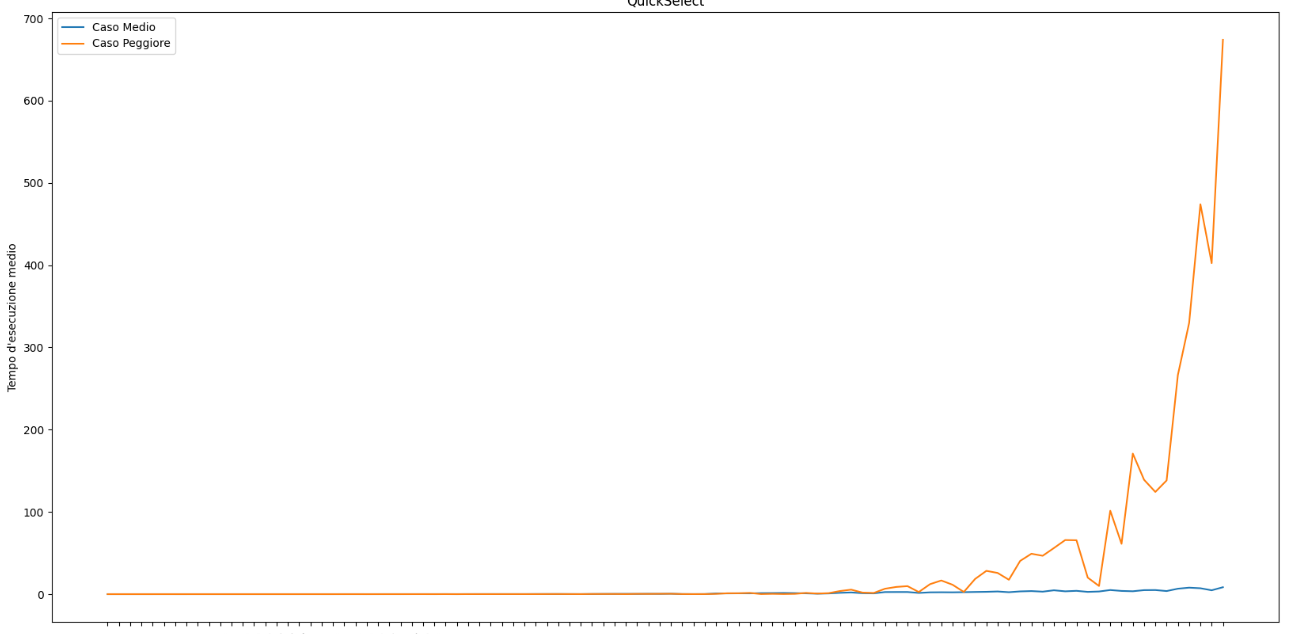
Ipotizziamo che richiamando ricorsivamente la funzione su quell’intervallo, partizioniamo ulteriormente l’array fino a trovare .

L’algoritmo e’ formalmente corretto.

Durante la fase di testing di QuickSelect, abbiamo misurato due esecuzioni con lo stesso array in input, ma con una gestione differente degli elementi contenuti al suo interno per simulare un caso medio e un caso peggiore. Nel caso medio, il k-esimo elemento da trovare e gli elementi nell’array vengono generati casualmente nel range di valori specificato dalla consegna. Tuttavia per il caso peggiore, si e’ deciso di sortare in ordine decrescente l’array (contenente gli stessi valori del caso medio), prima della misurazione dei tempi, per vedere come l’algoritmo si sarebbe comportato in ordine di tempo.

**Grafici**

Immagine che contiene testo, schermata, diagramma

Descrizione generata automaticamenteImmagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamente

Dai grafici notiamo che nel caso medio, il tempo di computazione rimane lineare, con un lieve aumento nella fase finale di testing. Per quanto riguarda il caso peggiore, il tempo di computazione inizia ad aumentare notevolmente quando la dimensione dell’array supera i 10.000 elementi, fino a toccare picchi da oltre 500 secondi.

**HeapSelect**

Lo scopo è quello di implementare un algoritmo di selezione che utilizzerà due min-heap, che denomineremo come H1 e H2. La prima heap sarà costituita a partire dal vettore fornito in input in tempo lineare e non verrà modificata. La seconda heap conterrà inizialmente un solo nodo, il quale corrisponderà alla radice di H1. All’i-esima iterazione, con *i* che andrà da a , l’algoritmo estrarrà la radice di H2 , che corrisponderà a un nodo xi in H1 e inserirà i nodi successori in H2. Dopo iterazioni, la radice di H2 corrisponderà al k-esimo elemento più piccolo del vettore in ingresso.

Inoltre, l’algoritmo dovrà avere una complessità di sia caso peggiore che nel caso medio.

**PseudoCodice**

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

**Complessità di HeapSelect**

|  |  |  |
| --- | --- | --- |
| Analisi del codice | **Costo nel caso medio** | **Costo nel caso peggiore** |
| **if**(*k****=****0* **or** *k* ***>*** **len**(*arr*)): return None |  |  |
| H1← A |  |  |
| heapify(H1) |  |  |
| H2 ← new\_arr() |  |  |
| heappush(H2,H1[0]) |  |  |
| i ← 0 |  |  |
| **while** (i **˂** k ): |  |  |
| (elemento, indice)← heappop(H2) |  |  |
| indice\_figlio\_sx ← 2 \* indice +1 |  |  |
| **if**(*indice\_figlio\_sx ˂ len(H1*) : |  |  |
| heappush(H2,H1[indice\_figlio\_sx]) |  |  |
| **elif** (*indice\_figlio\_dx ˂ len(H1*): |  |  |
| heappush(H2,H1[indice\_figlio\_dx]) |  |  |
| i ← i + 1 |  |  |

**Analisi dei casi:**

La complessità temporale di HeapSelect è strettamente influenzata dalla struttura della heap e dal numero di operazioni sulle heap. Questo perché ogni volta che viene rimosso un elemento diverso da una foglia, la heap dovrà essere riorganizzata per mantenere le sue proprietà.

* **Caso medio**: Il costo dell’inizializzazione delle strutture dati dipende strettamente da heapify, che ha complessita’ O(log(n)), con n la dimensione di H1. Tuttavia la costruzione del’heap ha costo O(n), perche’ piu’ della meta dei nodi si trova vicino alle foglie e quindi le operazioni richiedono un costo molto basso. Il ciclo while si ripete k volte e al suo interno vengono eseguite operazioni di inserimento e rimozione dalle heap. Il costo di Heappop e’ pari a log(m) con m la dimensione di H2. Il valore di m puo crescere fino a k.
* **Caso peggiore**: Il caso peggiore si verifica quando gli inserimenti richiedono il maggior tempo possibile per ripristinare le proprieta’ delle heap. L’inizializzazione delle strutture vale O(n) come nel caso medio, ma nel caso peggiore, se il valore di k aumenta, aumenta di conseguenza anche il costo di heappush, che ha costo nel caso peggiore pari a log(k)

**Correttezza di HeapSelect**

CASO BASE

Entriamo nel caso base di HeapSelect quando l’array A contiene un unico elemento. In questo caso, restituiamo l’unico elemento dell’array.

PASSO INDUTTIVO:

Ora passiamo all’analisi di HeapSelect nei 3 casi successivi al caso base, considerando che la lunghezza dell’array sia ora pari a n:

* Primo caso:

Dopo aver richiamato heapify sull’array, nel caso in cui l’indice del elemento è uguale a , verrà restituita la radice della nostra heap, concludendo cosi la procedura.

* Secondo caso:

Dopo la chiamata di heapify, se il valore di k è minore della metà della lunghezza dell’array, sapremmo che il elemento più piccolo si troverà nell sotto-albero sinistro. Ipotizziamo che richiamando ricorsivamente la funzione su quella sezione, troveremo l’elemento.

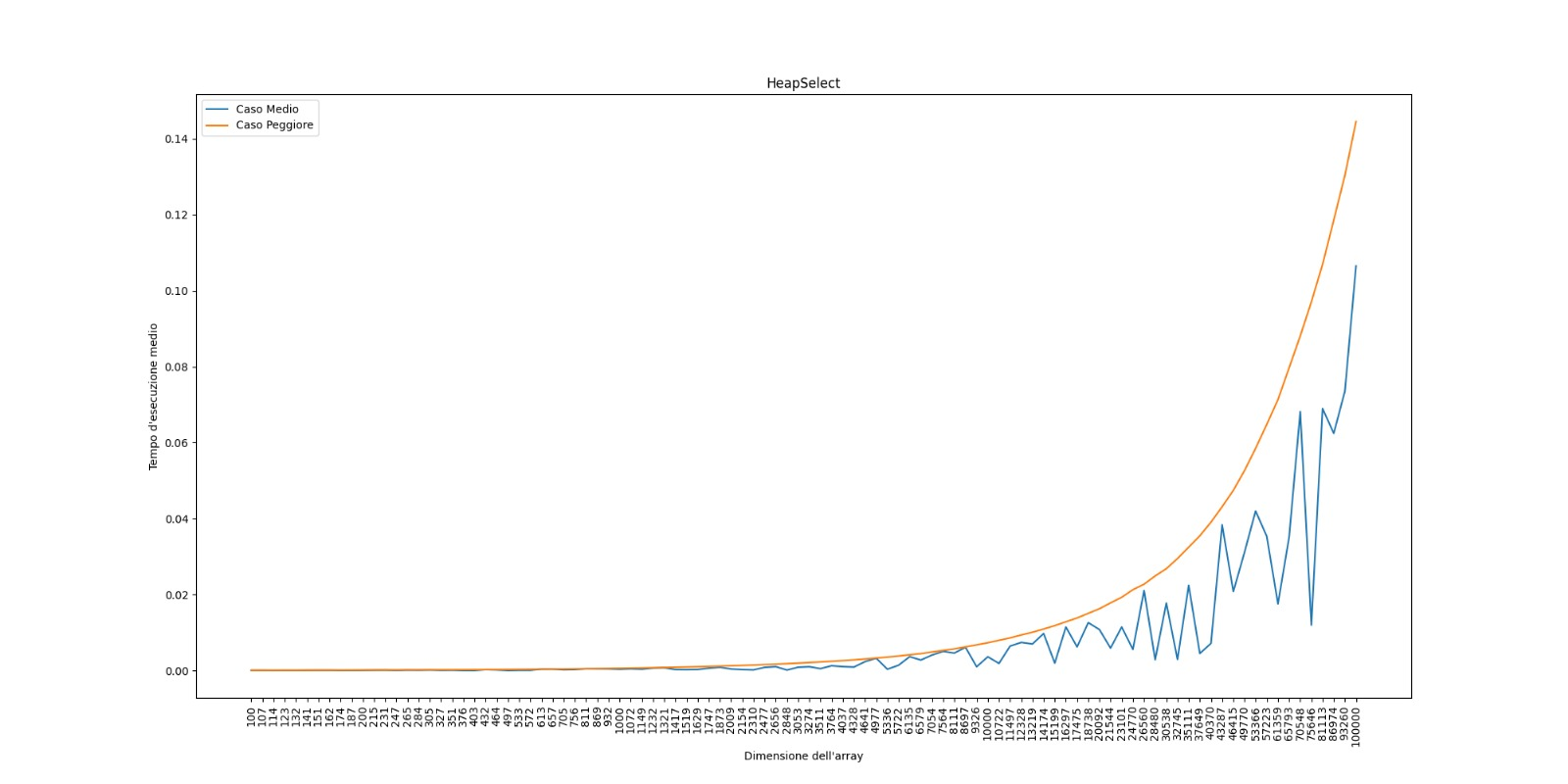
* Terzo caso:

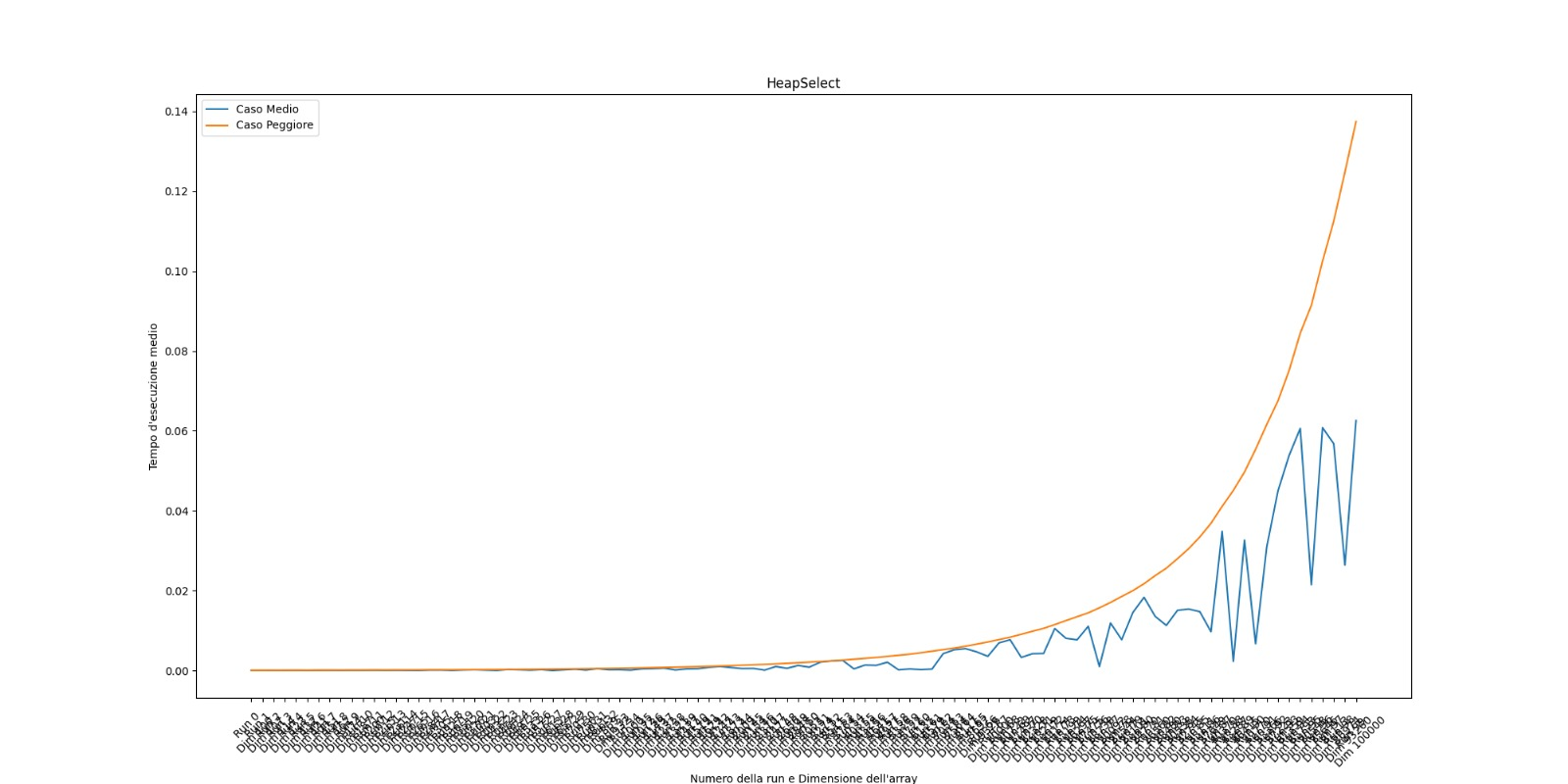
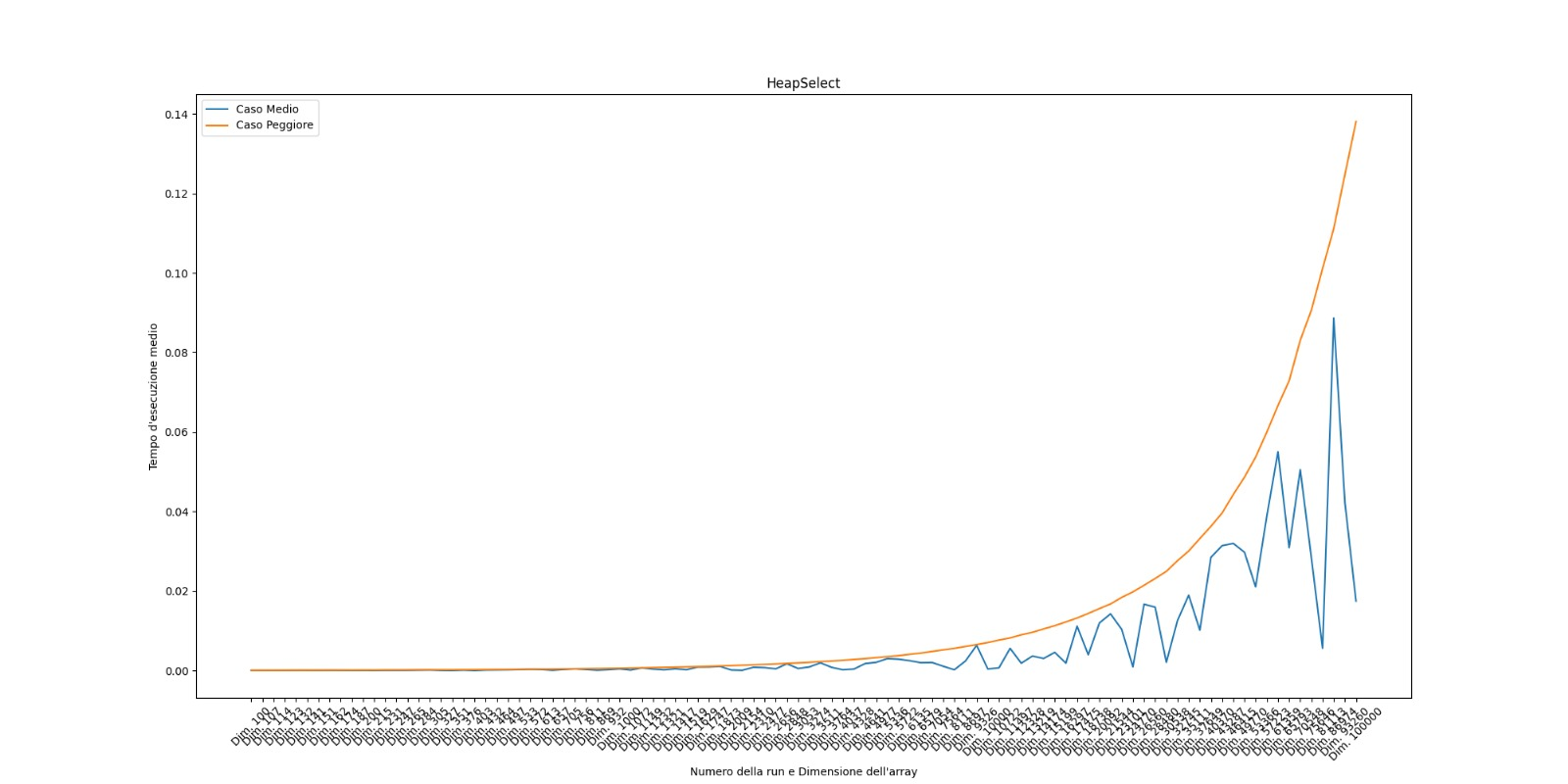
Dopo heapify, se il valore di k è maggiore della metà della lunghezza dell’array, sapremmo che il elemento più piccolo si trova nel sotto-albero destro. Ipotizziamo nuovamente che richiamando ricorsivamente la funzione su quella sezione, troveremo l’elemento.

L’algoritmo è formalmente corretto

**Grafici**

Durante la fase testing di HeapSelect, abbiamo misurato due esecuzioni con la stessa dimensione dell’array, ma con una generazione differente degli elementi contenuti al suo interno, in modo da poter simulare un caso medio e un caso peggiore, che sia significativamente distinguibile. Nel caso medio, gli elementi all’interno del nostro array e k, vengono generati in maniera casuale, nel range di valori specificato dalla consegna. Nel caso peggiore, gli elementi all’interno del nostro array vengono generati per formare una sequenza di valori minori e maggiori alternati, per complicare ulteriormente il lavoro di inserimento nell’heap.





Dai grafici notiamo che nel caso peggiore, il tempo di computazione cresce un po di piu’ rispetto al caso medio, perche’ l’algoritmo si concentra principalmente sul ripristino delle proprieta delle heap. Il tempo massimo di computazione del caso medio e’ di circa 10 ms, mentre per quanto riguarda il caso peggiore, circa 15 ms.

**Median of Medians Select**

Lo scopo è quello di implementare un algoritmo basato sulla suddivisione del vettore fornito in input in blocchi di dimensione limitata e calcolarne la mediana delle mediane. L’algoritmo eseguirà le seguenti operazioni:

-Divisione dell’array in blocchi di , escluso l’ultimo blocco, il quale potrebbe contenere meno di 5 elementi

-Ordinamento e calcolo della mediana di ogni blocco generato dal punto precedente

-Calcolo della mediana M delle mediane ricavate precedentemente, attraverso chiamata ricorsiva allo stesso algoritmo

-Partizionamento dell’intero array attorno alla mediana M, attraverso una variante di Partition

-Chiamata ricorsiva nella parte di array che sta a sinistra o a destra della mediana M, in funzione del valore fornito in input.

Inoltre, l’algoritmo dovrà avere una complessità nel caso peggiore, sia temporale che spaziale, pari a

**PseudoCodice**

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Immagine che contiene testo, schermata, documento, Carattere

Descrizione generata automaticamente

**Complessità di Median of Medians**

|  |  |
| --- | --- |
| Analisi del codice | **Costo nel caso peggiore** |
| if k < 0 or k >= len(arr) then  return None  end if |  |
| if l == r then  return arr[l]  end if |  |
| i = 0  mediani = [] |  |
| partizioni = (r - l + 1) // 5 |  |
| if (r - l + 1) % 5 != 0 then  partizioni = partizioni + 1  end if |  |
| while i < partizioni do |  |
| indice\_sx\_partizione = l + i \* 5 |  |
| indice\_dx\_partizione = min(l + i \* 5 + 4, r) |  |
| mediani.append(mediano(arr, indice\_sx\_partizione, indice\_dx\_partizione)) |  |
| i = i + 1  end while |  |
| if len(mediani) == 1 then |  |
| mediano\_dei\_mediani = mediani[0] else |  |
| mediano\_dei\_mediani = MedianOfMedians(mediani, 0, len(mediani) - 1, len(mediani) / 2  end if |  |
| indice\_pivot = partizione(arr, l, r, index(mediano\_dei\_mediani)) |  |
| if k == indice\_pivot then  return arr[k] |  |
| else if k < indice\_pivot then |  |
| return MedianOfMedians (arr, l, indice\_pivot - 1, k) |  |
| return MedianOfMedians (arr, indice\_pivot + 1, r, k)  end if |  |

Il procedimento all’inizio della procedura richiede un tempo di O(n) per il calcolo del numero di partizioni da usare e per trovarne il mediano, perche’ il ciclo while viene ripetuto al massimo n/5 volte e quindi la sua complessita e’ O(n). La chiamata ricorsiva per trovare il mediano dei mediani richiede un costo di . L’algoritmo infine fa una chiamata ricorsiva su una delle due porzioni dell’array, per un costo massimo pari a .

**Correttezza di MedianOfMedians**

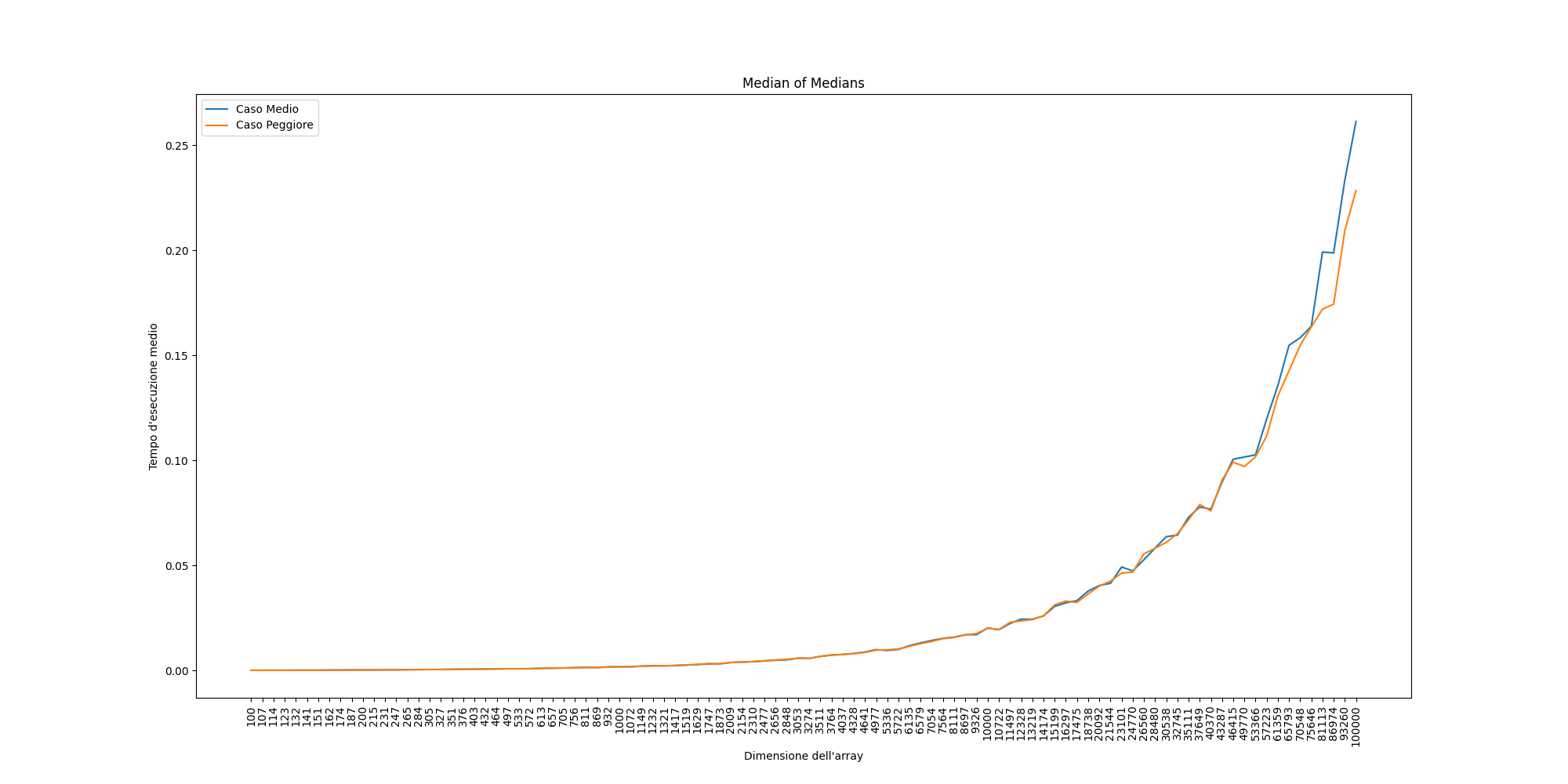
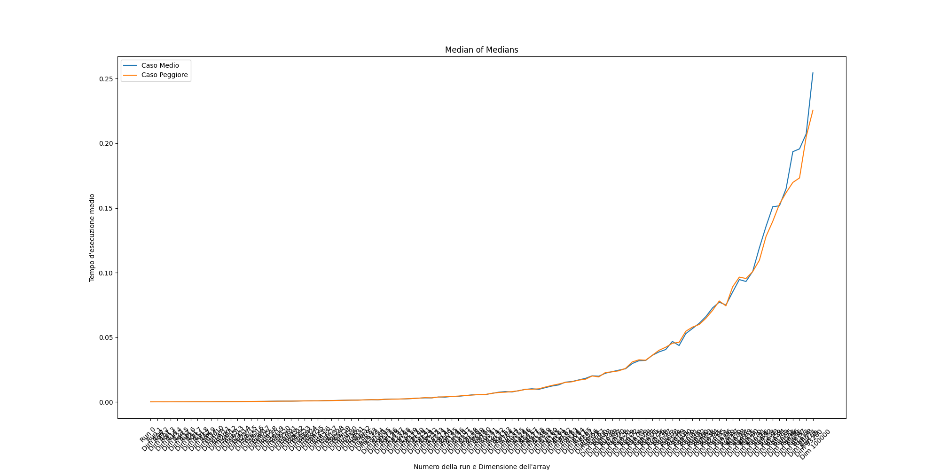
CASO BASE Entriamo nel caso base di MedianOfMedians quando l’array A contiene un unico elemento. In questo caso, restituiamo l’unico elemento dell’array.   
  
PASSO INDUTTIVO:

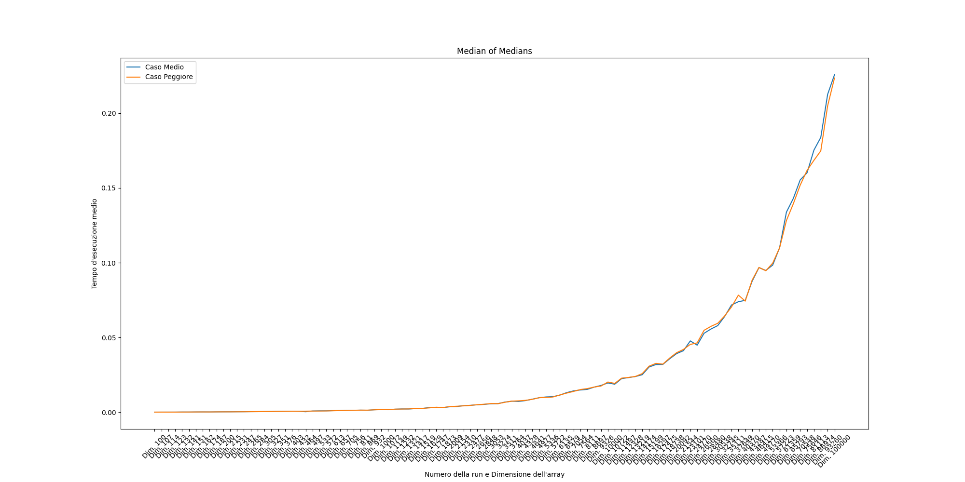
Ora passiamo all’analisi di MedianOfMedians nei 3 casi successivi al caso base, considerando che la lunghezza dell’array sia ora pari a n:

* Primo caso:   
  Dopo aver partizionato l’array, se l’indice del pivot è uguale a k, restituiamo l’elemento in posizione k.  
    
  • Secondo caso:

Dopo la partizione, se k è minore dell’indice del pivot, sappiamo che il k-esimo elemento più piccolo si trova nell’intervallo che va da l a pivot - 1. Ipotizziamo che richiamando ricorsivamente la funzione su quell’intervallo, troviamo l’elemento.  
  
• Terzo caso:   
Dopo la partizione, se k è maggiore dell’indice del pivot, sappiamo che il k-esimo elemento più piccolo si trova nell’intervallo che va da pivot + 1 a r. Ipotizziamo che richiamando ricorsivamente la funzione su quell’intervallo, troviamo l’elemento.

**Grafici**

****

****

Dal grafico notiamo come la complessita in ordine di tempo, di caso medio e caso peggiore di Median of medians rimanga sostanzialmente molto simile.

**Algoritmo di misurazione dei tempi d’esecuzione**

Per misurare i tempi d’esecuzione degli algoritmi e ricavare i grafici, abbiamo usato strutture dati come i dizionari, per salvare i tempi nei vari casi e le dimensioni degli array. Per ogni array generato, abbiamo testato ogni algoritmo per il caso medio e successivamente modificato l’array per ricreare i casi peggiori a seconda dell’algoritmo utilizzato. Il codice di questo programma si trova su GitHub al link posto nei riferimenti sottostanti. Sempre nella stessa repo si trovano i vari file .py usati per generare i vettori a seconda dell’esigenza, oltre che l’implementazione dei vari algoritmi e i grafici in .png

**Riferimenti**

* **Versione Python**: 3.12.2
* **Hardware**:
  + OS: Windows 10 Pro 64-bit
  + CPU: Amd Ryzen 7 4800H
  + RAM: 16GB DDR4-3200
* **Studenti**:
  + Fabio Pistrin | 162175 | 162175@spes.uniud.it
  + Elia Chiandussi | 162458 | 162458@spes.uniud.it
  + Stan Robert Gabrielle | 157077 | 157077@spes.uniud.it
  + Matteo Bovo| 163658 | 163658@spes.uniud.it
* GitHub:

<https://github.com/robert-stan/Progetto-ASD.git>