PRIMA PROVA PRATICA IN ITINERE

Ingegneria degli Algoritmi 2018/2019

o o o

Progetto 2 [*Pepè Sciarria*]

o o o

RELAZIONE DI:
Adriano Bramucci (Mat. 0240128)
Matteo Conti (Mat. 0244242)
Stefano Picchioni (Mat. 0218935)

Nel Progetto 2 è richiesta l'implementazione di un algoritmo in grado di selezionare il mediano di un sottoinsieme di elementi dell'input, da poter utilizzare come pivot nell'algoritmo di ordinamento **quickSort**.
Questo modo di scegliere l'elemento si basa sull'idea dei due informatici Robert W. Floyd e Ronald L. Rivest i quali osservarono che una scelta più accurata del pivot, preso come k-esimo elemento di un opportuno campione, restituisce un valore più vicino alla k-esima posizione effettiva producendo notevoli vantaggi in termini di efficienza.
L'algoritmo richiesto denominato **sampleMedianSelect** nasce, quindi, da una generalizzazione della soluzione appena descritta.
In particolare:

- sceglie un sottoinsieme V di m elementi in modo random;
- seleziona il mediano di V e lo usa come pivot.

In questa relazione si andrà a commentare la scelta delle soluzioni implementative adottate nella realizzazione dell'algoritmo ed il funzionamento dello stesso, ponendo particolare attenzione al modo in cui esso lavora in funzione dei parametri ricevuti.

Nel corso dell'analisi si andrà a confrontare l'efficienza della soluzione proposta rispetto agli altri **algoritmi di ordinamento studiati**, in funzione della dimensione dell'input e della cardinalità **m** del sottoinsieme **V**.

Funzionamento sampleMedianSelect

L'algoritmo da noi sviluppato sampleMedianSelect riceve come input l'array su cui operare, eventuali parametri per l'inizio e fine della lista, e la dimensione del sottoinsieme.

PseudoCodice SampleMedianSelect:

```
Algoritmo sampleMedianSelect (array, interi i, j, m)
  if m >= lunghezza della Lista do
     while m > lunghezza dell'array do
          Decremento m
  V <- Creo Sottoinsieme di dimensione m di elementi casuali dell'array
  if m <= 20 do
       Ordino il sottoinsieme con Insertion, Selection o BubbleSort
  else Ordino tramite Algoritmo a scelta
     Alternativamente:
       Seleziono il mediano di V con un algoritmo di selezione a scelta
  return mediano di V
```

Come si evince dallo pseudocodice l'algoritmo esegue un passo base per controllare se la dimensione del sottoinsieme è maggiore o uguale della dimensione dell' array, nel caso in cui fosse vero decrementa opportunamente m in modo che in una possibile applicazione pratica l'algoritmo non termini inutilmente. Concluso il passo base crea un sottoinsieme di dimensione m di elementi casuali dell'array in ingresso e in base al valore di m esegue l'algoritmo di ordinamento appropriato:

- M minore di 20: SelectionSort, InsertionSort o BubbleSort in quanto come visto a lezione se devono ordinare un insieme piccolo risultano essere più prestanti di algoritmi più complessi.
- M maggiore di 20: HeapSort, MergeSort ecc.

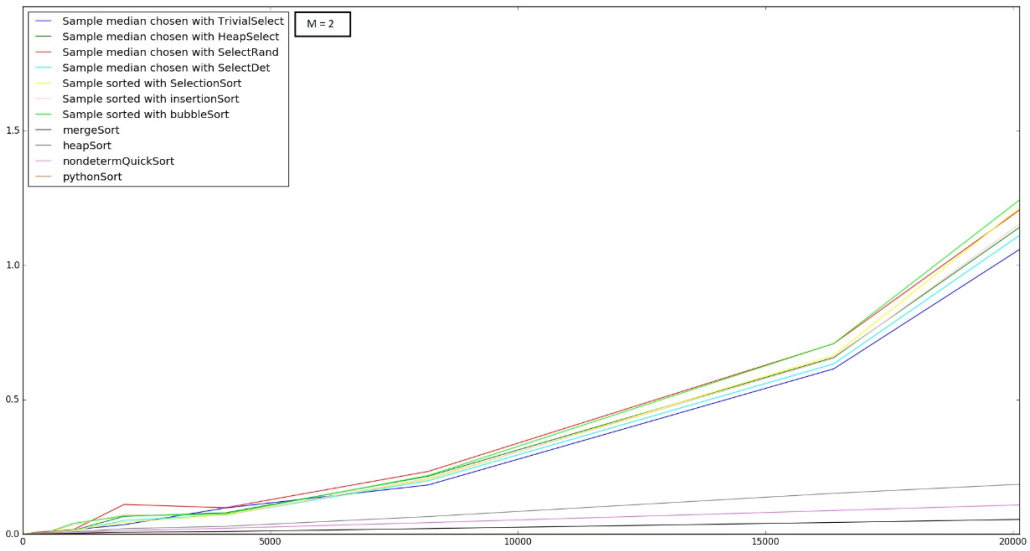
Oppure

Dopo aver scelto il sottoinsieme V di elementi dell'array iniziale ne viene selezionato il mediano tramite un algoritmo di selezione a scelta tra quelli visti a lezione:

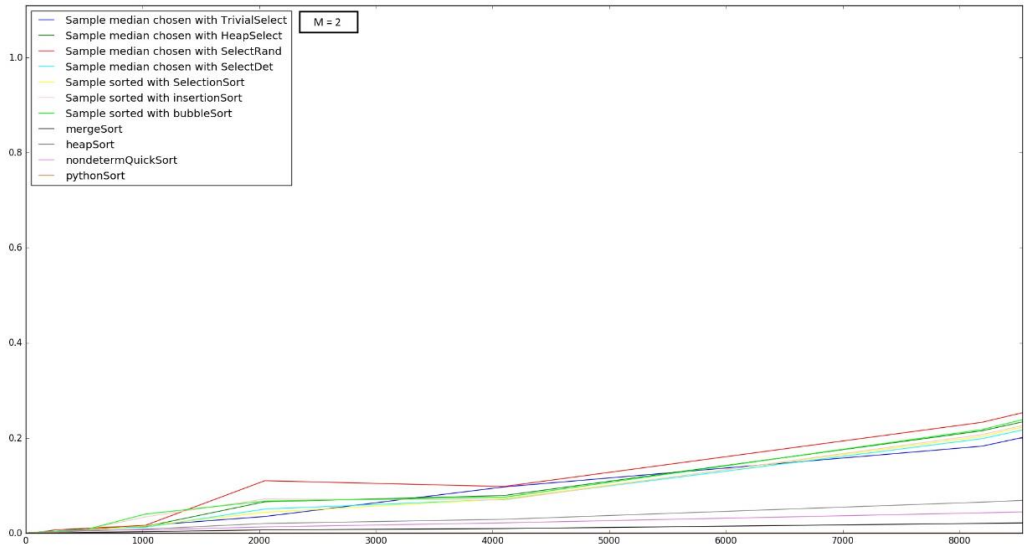
- SelectDet,SelectRand,HeapSelect,TrivialSelect.

Infine, ritorna il mediano del sottoinsieme ordinato.

Risultati per m = 2 :



↑ Grafico generale (m=2)



↑ Grafico ristretto (m = 2 , n < 8000)

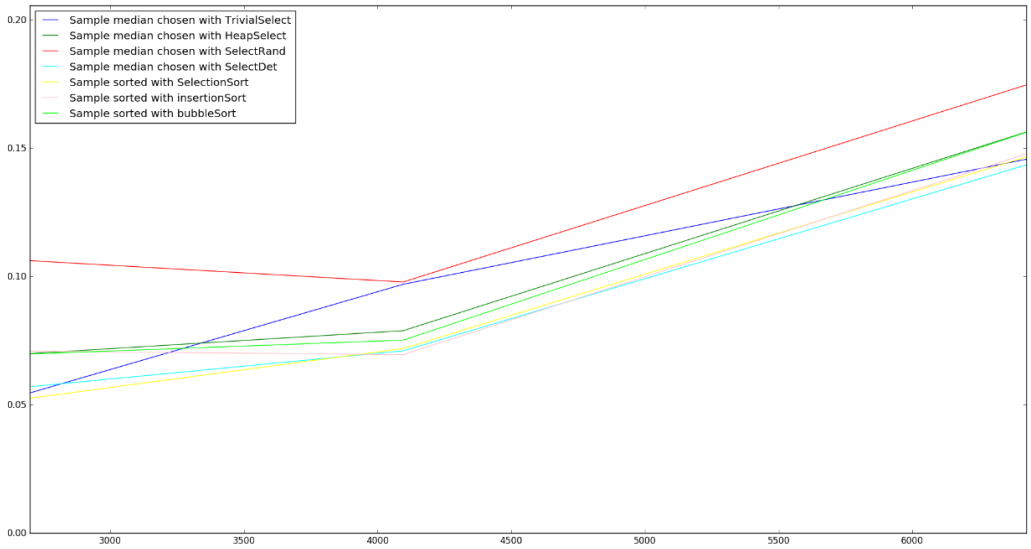
RISULTATI SAMPLEMEDIANSELECT:

In questo caso l'algoritmo ricade nella selezione del mediano effettuando prima l'ordinamento dell'array, tramite insertionSort, selectionSort o bubbleSort, e quindi ritornando il valore corretto.

- Per una dimensione dell'input molto contenuta la differenza tra i 3 è davvero minima, e rimane tale fino al valore di n ~ 500 .
- Da quel valore la situazione è più marcata a favore del selectioSort che risulta la migliore soluzione a scapito dell'insertionSort ed in ultimo il bubbleSort.

- Da una dimensione dell'input n~10000 invece l'insertionSort risulta la soluzione ottimale, mentre la bubbleSort rimane la soluzione meno efficiente.

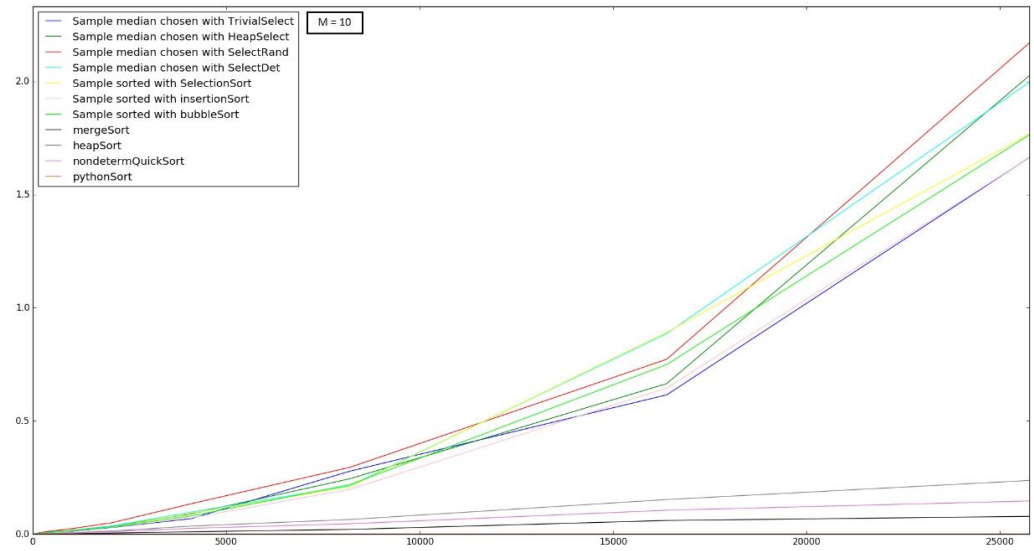
TEST VARIANTI SELECTION NON UTILIZZATE (in SAMPLEMEDIANSELECT):
Testando comunque gli algoritmi di selezione all'interno della soluzione proposta sampleMedianSelect otteniamo che per input elevati (n~1400) l'uso della selectRand per il mediano è nettamente la soluzione peggiore, mentre la migliore è la trivialSelect.
- Per input contenuti (n<50) la soluzione più efficiente invece è la selectDet, quindi la trivialSelect(che però ha un picco peggiore per n~4), mentre la peggiore è l'heapSelect.
La soluzione migliore (anche se non prevista dall'algoritmo per m=2) è quella che utilizza il trivialSelect (nonostante dei rallentamenti per 3000<n<6000)



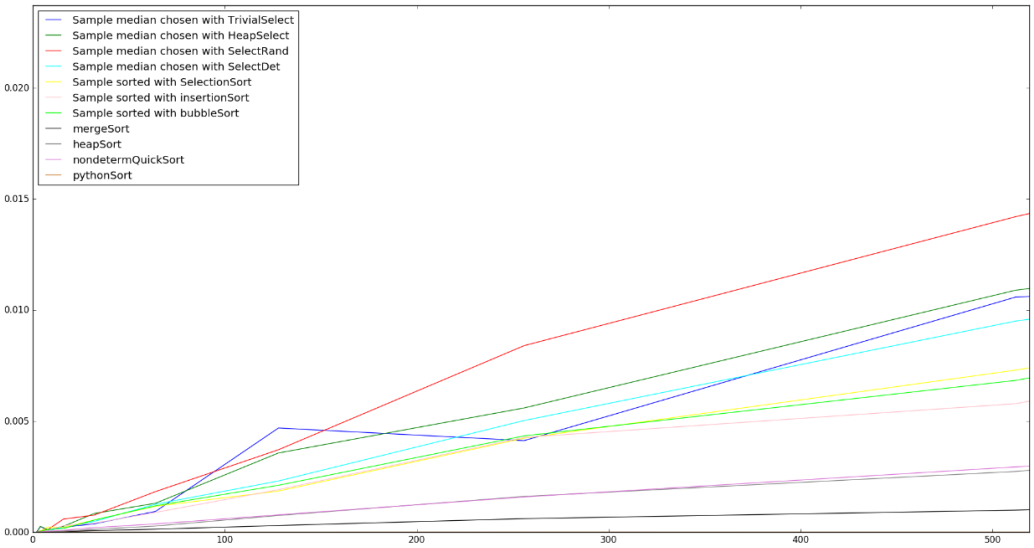
↑ calo efficienza trivialSelect (m = 2 , 3000 < n < 6000)

CONFRONTO GENERALE:
Da un'analisi generale invece tra tutti gli algoritmi le soluzioni più efficienti per l'ordinamento risultano essere in ordine il pythonSort, il mergeSort, il quickSort classico, e l'heapSort, con quest'ultimo migliore rispetto al predecessore fino ad un input n~650.

Risultati per m = 10 :



↑ Grafico generale (m=10)



↑ Grafico ristretto (m = 10 , n < 500)

RISULTATI SAMPLEMEDIANSELECT:

In questo caso l'algoritmo ricade ancora nella selezione del mediano effettuando prima l'ordinamento dell'array, tramite insertionSort, selectionSort o bubbleSort, e quindi ritornando il valore corretto.
- Per input di dimensioni n<10 la soluzione più efficiente ricade sulla selezione del mediano ordinando con l'insertionSort.
- Per dimensioni dell'input 10<n<~50 la scelta migliore per l'algoritmo rimane l'insertionSort, mentre la peggiore è il bubbleSort.

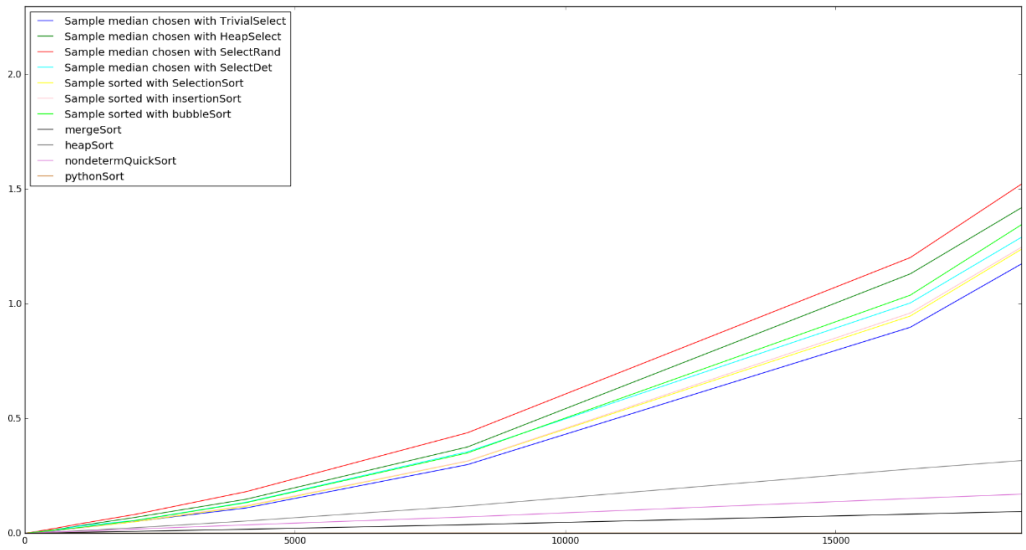
- Per dimensioni n~500 la soluzione migliore sfrutta ancora l'insertionSort, ma stavolta il caso peggiore è col selectionSort.
- Per dimensioni ancora crescenti la variante consigliata rimane quella che sfrutta l'insertionSort, che è quella più accreditata per lavorare efficientemente con questa dimensione del sottoinsieme V.

TEST VARIANTI SELECTION NON UTILIZZATE (in SAMPLEMEDIANSELECT):
- Per input di dimensioni n<10 come già detto, la soluzione più efficiente per la selezione ricade sull'uso di insertionSort, ma confrontando il risultato con gli algoritmi di selezione anche utilizzare la selectDet genera risultati soddisfacenti.
- Per dimensioni dell'input 10<n<~50 l'utilizzo di algoritmi di selezione per il mediano sono più indicate poiché l'uso del trivialSelect ha un andamento di poco peggiore all'insertionSort, ma più lineare, costante, eccetto per la dimensione analizzata precedentemente (n<10) dove risulta uno dei peggiori. La variante più inefficiente è di gran lunga quella che fa uso della selectRand.
- Per dimensioni n~500 la variante di selezione migliore è la selectDet, la peggiore la selectRand.
- Per dimensioni molto grandi dell'input la soluzione migliore tra le varianti select risulta la trivialSelect, nonostante è più consigliato l'uso dell'ordinamento tramite insertionSort in quanto più uniforme.

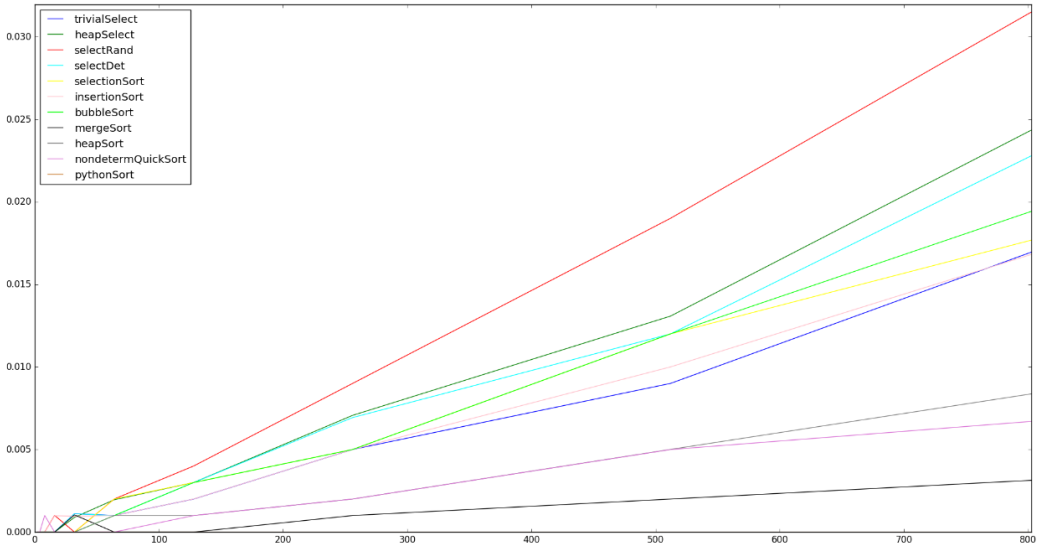
CONFRONTO GENERALE:

Da un'analisi generale invece tra tutti gli algoritmi le soluzioni più efficienti per l'ordinamento, come prima, risultano in ordine il pythonSort, il mergeSort, il quickSort classico, e l'heapSort (che per n<~600 lavora leggermente meglio rispetto al predecessore).

Risultati per m = 20 :



↑ Grafico generale (m=20)



↑ Grafico ristretto (m=20, n<800)

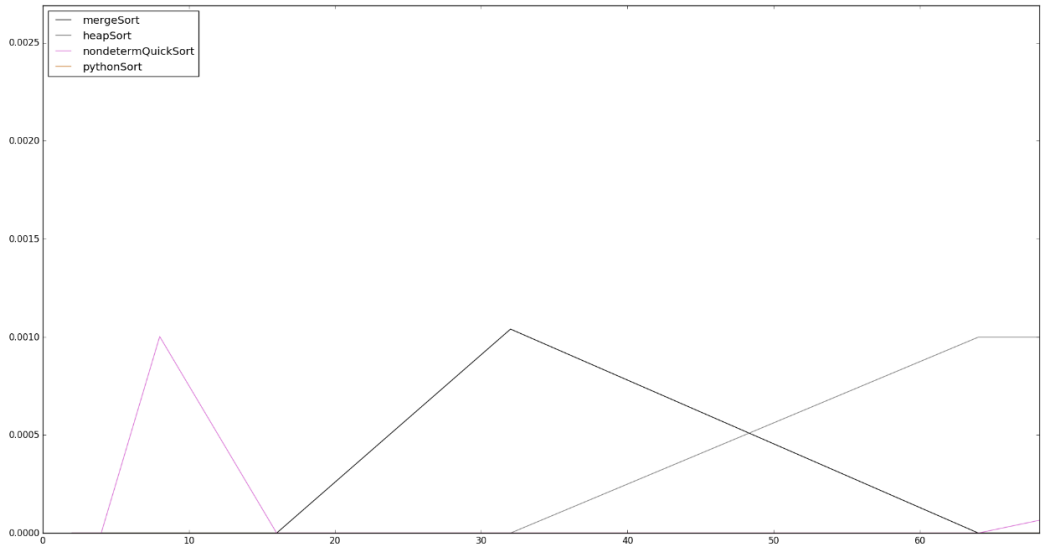
RISULTATI SAMPLEMEDIANSELECT:

In questo caso l'algoritmo proposto abbandona gli algoritmi di ordinamento più semplici per lavorare con quelli più complessi o quelli di selezione.

- Per input molto grandi i risultati sono abbastanza lineari con una netta preferenza per l'uso del trivial select, mentre la peggiore risulta la selectRand.
- Per input piccoli (n fino ~150) i risultati sono molto caotici tuttavia è ancora consigliato l'uso del trivialSelect, leggermente migliore sugli altri.

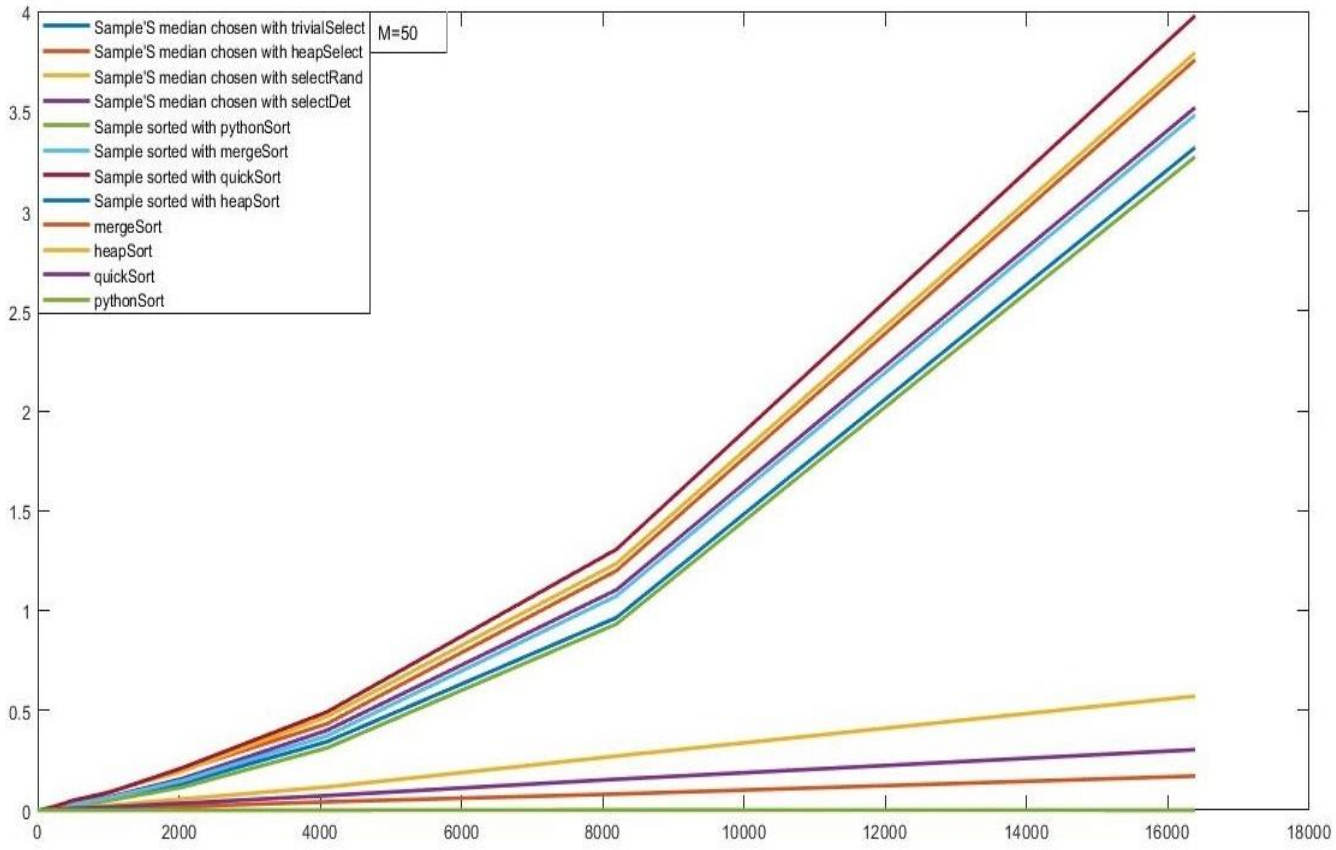
TEST VARIANTI SORT (in SAMPLEMEDIANSELECT):
Tra le varianti non utilizzate spicca comunque l'utilizzo del selectionSort per la ricerca del mediano che risulta leggermente meno efficiente della trivialSelect consigliata precedentemente.
Mentre la variante peggiore, tra queste, è la selezione tramite l'uso della bubbleSort.

CONFRONTO GENERALE:
Anche in questo caso ordinare direttamente l'array risulta un'operazione più vantaggiosa, con il solito pythonSort a dominare su mergeSort, quickSort classico ed heapSort.
Notare come il quickSort classico abbia un picco per valori di input piccoli.



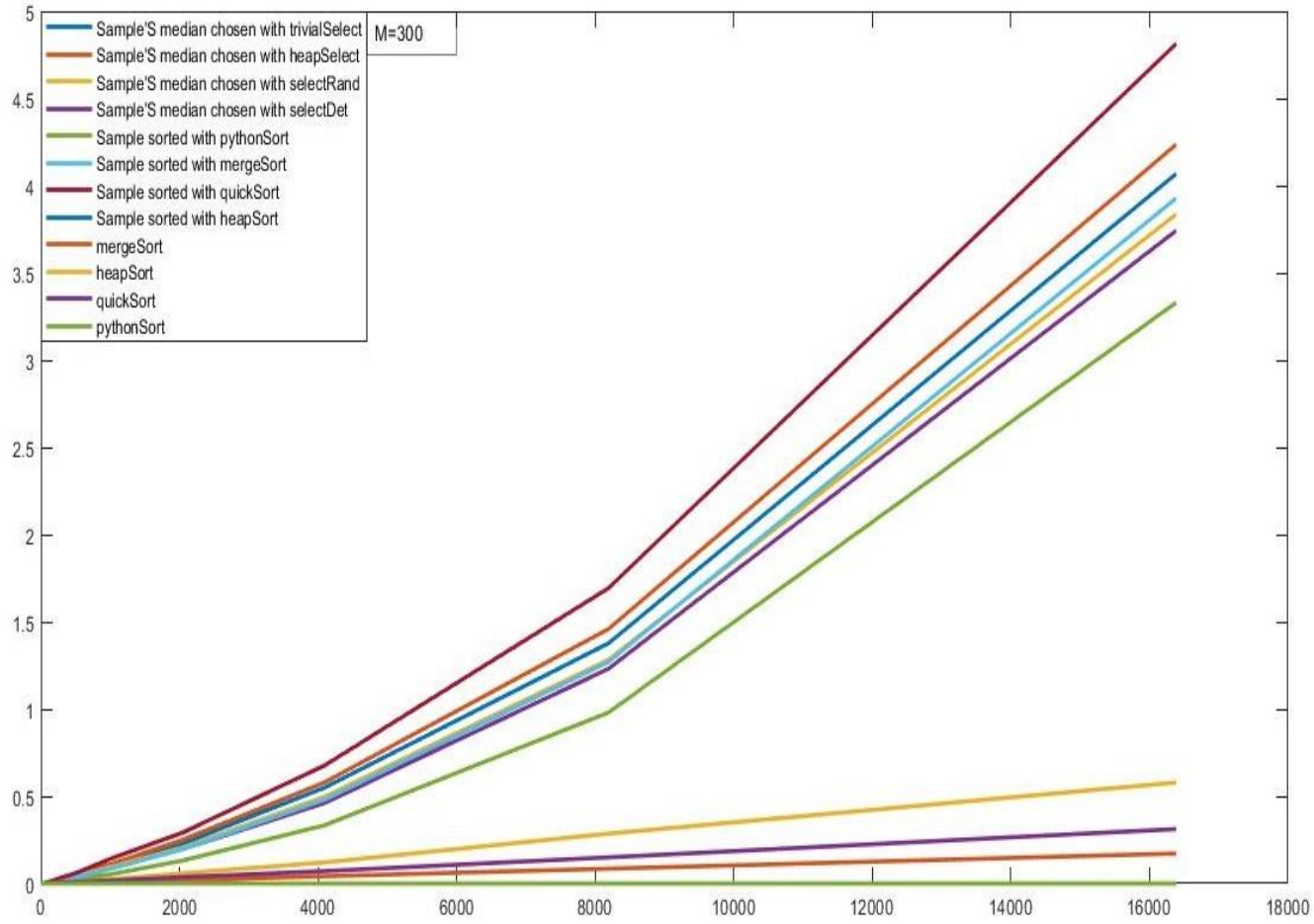
↑ quickSort non-det per piccoli input (m=20)

Risultati per m = 50 :



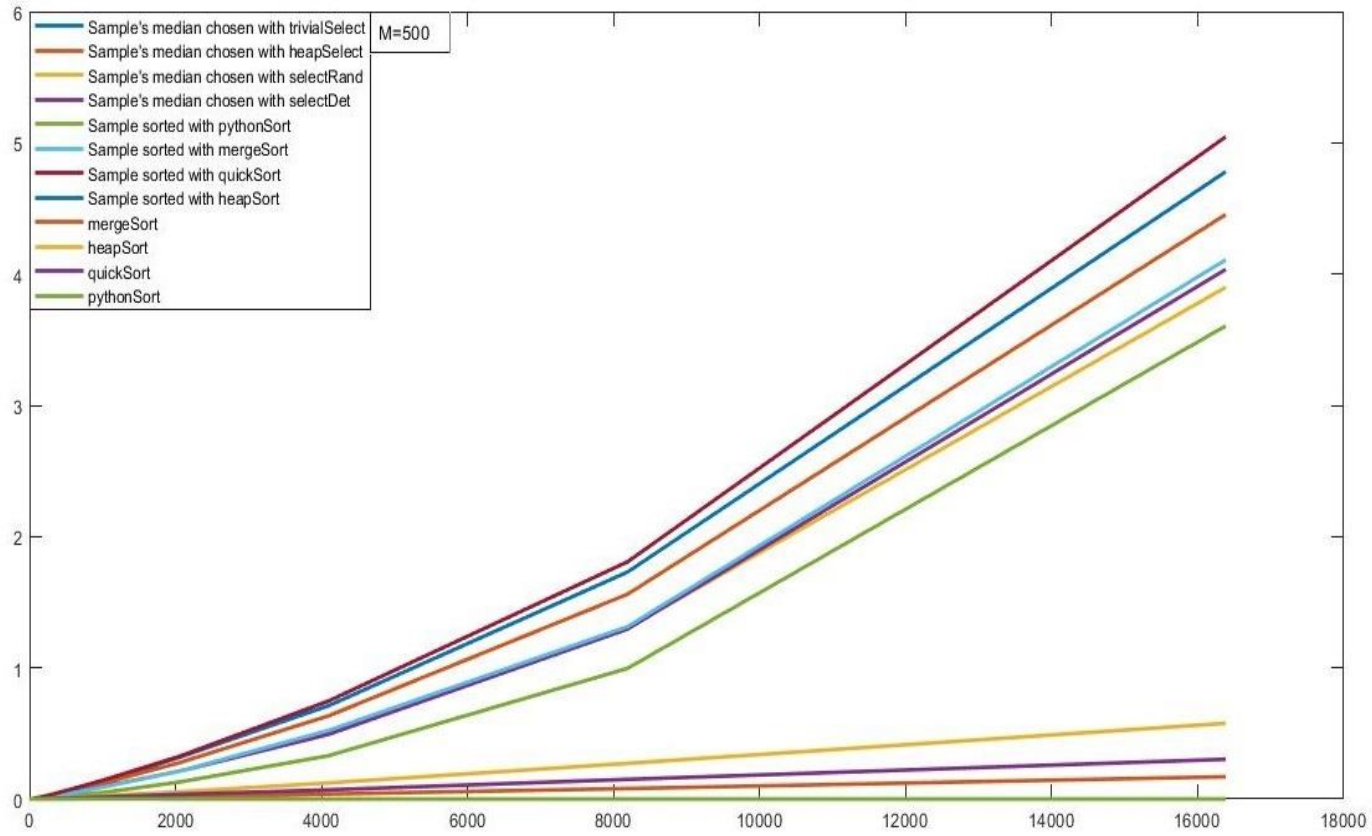
Come si può vedere dal grafico il quickSort che sceglie il pivot con sampleMedianSelect (qualsiasi variante)si dimostra al crescere di n di gran lunga più lento degli altri algoritmi visti a lezione prendendo un andamento simil-parabolico fin da subito producendo dei tempi di esecuzione dell'ordine del secondo gia da array di 7000-8000 elementi. Dal grafico si può osservare che la variante migliore risulta essere quella in cui il sottoinsieme di dimensione m dell'array viene ordinato tramite il metodo Sort di python e successivamente ne viene restituito l'elemento centrale, il peggiore risulta essere la variante che utilizza il quickSort classico per ordinare il campione e prenderne l'elemento centrale. Si può notare anche come a differenza dei casi precedenti tutte le varianti di quickSort abbiano un andamento abbastanza simile e regolare tra di loro. Una cosa curiosa è che tranne la variante con il quickSort classico le altre varianti sono molto vicine a coppie:
heapSelect-selectRand,selectDet-mergeSort,pythonSort-heapSort

Risultati per m = 300 :



Le considerazioni generali sul caso in cui m sia 300 sono simili a quelle per m=50 in quanto come si può notare il grafico è molto simile, si può notare però che come era logico aspettarsi che all'aumentare di m diminuiscono le prestazioni infatti si arriva a sfiorare il secondo nel tempo di esecuzione già da array di 5000-6000 elementi. Un fenomeno particolare è che si fa più marcata la differenza tra il migliore (campione ordinato con Sort di python) ed il peggiore (campione ordinato con quickSort classico) mentre le varianti con un tempo intermedio tra i due casi sopracitati rimangono ora più vicine tra loro come tempistiche.

Risultati per m = 500



Come prima considerazioni generali sul caso in cui m sia 500 sono simili a quelle per m=50,300 in quanto come si può notare il grafico è molto simile, si nota inoltre che però la soglia del secondo si è scesa ma non di troppo e si assesta sempre intorno ad n = 5000. La variante migliore e peggiore rimangono uguali tuttavia quelle intermedie hanno un gap di prestazione omogeneo tra di loro eccetto per 3 che sono molto simili cioè selectRand-selectDet-mergeSort.

Casi particolari:

Sono stati effettuati dei test su input particolari come un array vuoto, un array con stessi elementi, un array ordinato ed uno ordinato al contrario.
In riferimento ai grafici presenti nella cartella "casi_particolari", possiamo affermare che per input estremi (caso vuoto e con stessi elementi) l'efficienza migliora ovviamente in maniera notevole. Nel caso di input ordinati ciò non accade così esplicitamente, otteniamo solo un lieve miglioramento per n grande. Infine nel caso di input ordinati al contrario, per quanto riguarda array di dimensioni molto elevate possiamo trovare un lieve miglioramento, mentre per valori più bassi (in particolare 1000<n<10000) l'efficienza risulta peggiore rispetto ad un caso comune(random).

Conclusione

Dopo l'esecuzione dei vari algoritmi abbiamo raccolto abbastanza informazioni sulle tempistiche e l'efficienza, infine, abbiamo riportato il tutto in tabella confrontando le varianti del quickSort da noi implementato con gli altri algoritmi di ordinamento visti a lezione, limitando N ed M a dei casi intermedi.

N = 2 ed M piccoli

Sample Trivial Select	Sample Heap Select	Sample Select Rand	Sample Select Det	Sample Selection Sort	Sample Insertion Sort	Sample Bubble Sort	Merge Sort	Heap Sort	Classic QuickSort	Python Sort	m
0,2ms	0,2ms	0,07ms	0,04ms	0,05ms	0,04ms	0,05ms	0,09ms	0,02ms	0,02ms	0,002ms	2
0,05ms	0,04ms	0,06ms	0,03ms	0,03ms	0,03ms	0,03ms	0,006ms	0,01ms	0,02ms	0,0008ms	10
0,05ms	0,04ms	0,05ms	0,04ms	0,03ms	0,03ms	0,05ms	0,005ms	0,01ms	0,02ms	0,0009ms	20

N = 512 ed M piccoli

Sample Trivial Select	Sample Heap Select	Sample Select Rand	Sample Select Det	Sample Selection Sort	Sample Insertion Sort	Bubble Sort	Merge Sort	Heap Sort	Classic QuickSort	Python Sort	M
10ms	12ms	15ms	13ms	15ms	13ms	15ms	2ms	6ms	6ms	0,02ms	2
13ms	17ms	23ms	15ms	13ms	13ms	14ms	2ms	7ms	5ms	0,01ms	10
13ms	18ms	24ms	17ms	14ms	17ms	16ms	2ms	7ms	5ms	0,01ms	20

N = 8192 ed M piccoli

Sample Trivial Select	Sample Heap Select	Sample Select Rand	Sample Select Det	Sample Selection Sort	Sample Insertion Sort	Sample Bubble Sort	Merge Sort	Heap Sort	Classic QuickSort	Python Sort	M
594,2ms	650ms	696ms	628ms	653ms	648ms	703ms	43ms	137ms	87ms	2ms	2
601ms	667ms	769ms	630ms	630ms	622ms	662ms	43ms	137ms	88ms	2ms	10
620ms	698ms	781ms	660ms	649ms	641ms	687ms	43ms	137ms	88ms	2ms	20

N = 2 ed M grandi

Sample Trivial Select	Sample Heap Select	Sample Select Rand	Sample Select Det	Sample Merge Sort	Sample Heap Sort	Sample Quick Sort	Sample Python Sort	Merge Sort	Heap Sort	Classic Quick Sort	Python Sort	M
0.07ms	0.06ms	0.06ms	0.002ms	0.06ms	0.05ms	0.05ms	0.06ms	0.002ms	0.02ms	0.002ms	0.001ms	50
0.1ms	0.09ms	0.1ms	0.09ms	0.08ms	0.08ms	0.08ms	0.08ms	0.002ms	0.02ms	0.002ms	0.0009ms	300
0.15ms	0.13ms	0.13ms	0.14ms	0.12ms	0.13ms	0.13ms	0.12ms	0.005ms	0.01ms	0.02ms	0.002ms	500

N = 512 ed M grandi

Sample Trivial Select	Sample Heap Select	Sample Select Rand	Sample Select Det	Sample Merge Sort	Sample Heap Sort	Sample Quick Sort	Sample Python Sort	Merge Sort	Heap Sort	Classic Quick Sort	Python Sort	M
29ms	37ms	34ms	27ms	27ms	36ms	39ms	21ms	2ms	6ms	5ms	0,01ms	50
44ms	51ms	49ms	45ms	44ms	55ms	61ms	36ms	2ms	6ms	5ms	0,01ms	300
65ms	67ms	62ms	63ms	60ms	73ms	60ms	58ms	2ms	6ms	5ms	0,01ms	500

N = 8192 ed M grandi

Sample Trivial Select	Sample Heap Select	Sample Select Rand	Sample Select Det	Sample Merge Sort	Sample Heap Sort	Sample Quick Sort	Sample Python Sort	Merge Sort	Heap Sort	Classic QuickSort	Python Sort	M
824ms	940ms	947ms	848ms	878ms	992ms	1.06s	777ms	43ms	139ms	88ms	0.2ms	50
1s	1s	1s	1s	1.18s	1.37s	1.44s	1.03m	43ms	138ms	87ms	0.2ms	300
1.6s	1.57s	1.44s	1,42s	1.48s	1.72s	1.78s	1.31s	42ms	138ms	88ms	0.2ms	500

Dai risultati ottenuti possiamo dedurre che gli algoritmi che interagiscono con la sampleMedianSelect lavorano meglio con un sottoinsieme di dimensioni ridotte (m piccoli), al contrario, con m grandi presentano tempi maggiori agli algoritmi tradizionali visti a lezione, in generale le versioni
Come conclusione possiamo affermare che il nostro algoritmo non è abbastanza efficiente da superare gli algoritmi tradizionali.

Moduli del Progetto

Cartella project_test_plot

1- **SelectSampleMedianSelect.py:** Il modulo contiene il codice dell'algoritmo **quickSort** in cui la scelta del pivot avviene tramite l'algoritmo **sampleMedianSelect** il quale sceglie il mediano del campione tramite un algoritmo di selezione.

2 - **SortSampleMedianSelect.py:** Il modulo contiene il codice dell'algoritmo **quickSort** in cui la scelta del pivot avviene tramite l'algoritmo **sampleMedianSelect** il quale sceglie il mediano del campione ordinandolo e restituendo l'elemento centrale.

3 – **demoQuickSort.py:** Il modulo contiene le varie prove eseguite con i diversi algoritmi adottati. Inoltre, ci consente di osservare i vari tempi ottenuti.

4 – **SelectionSort.py:** Il modulo contiene il codice relativo all'algoritmo selectionSort visto a lezione.

5 – **InsertionSort.py:** Il modulo contiene il codice relativo all'algoritmo insertionSort visto a lezione.

6 – **BubbleSort.py:** Il modulo contiene il codice relativo all'algoritmo bubbleSort visto a lezione.

7 – **MergeSort.py**: Il modulo contiene il codice relativo all'algoritmo ricorsivo mergeSort dotato della funzione **merge**.

8 – **HeapSort.py**: Il modulo contiene il codice relativo all'algoritmo ricorsivo heapSort dotato della funzione **heapify**.

9 – **QuickSort.py**: Il modulo contiene il codice relativo all'algoritmo ricorsivo quickSort classico dotato della funzione **partition**.

10 – **quickSelectRand.py**: Il modulo contiene l'algoritmo di selezione **Select randomizzato** visto a lezione.

11 – **quickSelectDet.py**: Il modulo contiene l'algoritmo di selezione **Select deterministico** visto a lezione.

12 – **TrivialSelect.py**: Il modulo contiene il codice dell'algoritmo di selezione **trivialSelect** visto a lezione.

13 – **HeapMin.py/HeapMax.py**: I moduli contengono rispettivamente la struttura dati **heap** basata sul minimo e sul massimo necessari per l'algoritmo heapSelect ed heapSort.

14 – Vari **testXY.py** file: Contengono le modifiche apportate agli algoritmi di ordinamento, garantendo l'interazione con la sampleMedianSelect vista nel paragrafo precedente.

15 – **plotting.py**: questo modulo è una versione sintetizzata di demoQuickSort finalizzata al raccoglimento degli array di dati con i quali fare i grafici con pythonPlot e matlab.

16 – **plotM20.py**: questo modulo contiene il necessario per fare il grafico tramite python

Cartella matlab_graph

Questa cartella contiene il codice con il quale sono stati generati i grafici in matlab.

Cartella grafici

Questa cartella contiene i grafici inclusi nella relazione, di alcuni anche le versioni ingrandite, e quelli dei casi particolari testati.

Librerie Python:

random: funzione utilizzata per calcolare un numero casuale, dato un determinato range n.

ceil: funzione per l'approssimazione di un numero.

time: funzione per calcolare il tempo passato dall'esecuzione dell'algoritmo scelto.

Matplotlib: libreria utilizzata per fare i grafici.