Implementazione dell'algoritmo Timsort e confronto con gli algoritmi Mergesort e Quicksort

Marinella Negrini

July 25, 2020

1 Introduzione

1.1 Problema affrontato

Il lavoro svolto è incentrato sul problema algoritmico dell'ordinamento. Tale problema consiste nell'ordinare una sequenza di n elementi appartenenti ad un dominio in cui è definita una relazione d'ordine.
Per tale problema è nota una delimitazione inferiore pari a $\Omega(n \log n)$, quindi nessun algoritmo di ordinamento potrà impiegare un tempo inferiore a tale quantità.

1.2 Algoritmi considerati e obiettivi del progetto

Gli algoritmi di ordinamento su cui lo studio è incentrato sono: Mergesort, Quicksort e TimSort.

In particolare, un primo obiettivo è stato quello di produrre (nel linguaggio di programmazione Python 3.6 ed utilizzando PyCharm come IDE) le implementazioni per i tre algoritmi oggetto dello studio.
L'obiettivo successivo è stato quello di effettuare uno studio sperimentale sulle implementazioni realizzate, ponendo particolare attenzione al tempo impiegato da ognuno degli algoritmi per il calcolo della soluzione.

2 Descrizione delle implementazioni realizzate

2.1 Mergesort

Nell'implementazione realizzata sono eseguiti i passaggi dell'algoritmo, effettuando chiamate ricorsive alla funzione `MergeSort` sulle due metà bilanciate della sequenza di input e richiamando infine la funzione `Merge`, che estrae ripetutamente il minimo tra le due sequenze ordinate e lo pone nella sequenza in uscita, ottenendo un'unica sequenza ordinata.

Tenendo presente che la funzione `Merge` ha una complessità teorica pari a $\mathcal{O}(n)$, tale algoritmo ha una complessità che è espressa dalla seguente relazione di ricorrenza:

$$T(n) = 2T(\tfrac{n}{2}) + \mathcal{O}(n)$$

Di conseguenza la complessità teorica del Mergesort risulta essere $\mathcal{O}(n \log n)$, nel caso peggiore, migliore e medio.

L'implementazione realizzata è non in loco, infatti sono state utilizzate delle sequenze temporanee contenenti gli elementi delle sotto-sequenze ordinate da porre nella sequenza d'uscita.

Inoltre la funzione `Merge` è stata realizzata in modo da garantire la stabilità dell'algoritmo, mantenendo l'ordine relativo tra elementi con lo stesso valore (infatti nel caso di elementi uguali viene data la priorità alla sotto-sequenza di sinistra).

2.2 Quicksort

Nell'implementazione realizzata viene richiamata la funzione `Partition`, che sceglie il primo elemento della sequenza come "pivot" e pone gli elementi minori di esso alla sua sinistra e gli elementi maggiori alla sua destra (scambiando tra loro gli elementi che si trovano nella porzione sbagliata). Successivamente vengono effettuate chiamate ricorsive alla funzione `QuickSort` sulle due porzioni ottenute, non necessariamente bilanciate.

La funzione `Partition` ha una complessità pari a $\mathcal{O}(n)$. Il caso peggiore si ha per sequenze già ordinate, in quanto le due porzioni su cui viene richiamata ricorsivamente la funzione `QuickSort` sono sbilanciate; in tal caso tale algoritmo ha una complessità pari a $\mathcal{O}(n^2)$. Nel caso migliore (in cui si ottengono sempre porzioni perfettamente bilanciate) e nel caso medio, vengono vengono effettuate un numero logaritmico di chiamate ricorsive, quindi in tali casi la complessità teorica risulta essere $\mathcal{O}(n \log n)$.

L'implementazione realizzata è in loco, in quanto si opera tramite scambi di due elementi.

Inoltre non si ha stabilità, in quanto la funzione `Partition` non preserva l'ordine relativo tra elementi uguali.

2.3 Timsort

Per realizzare l'implementazione di tale algoritmo è stato fatto riferimento alla descrizione del suo funzionamento [1] e ad un'implementazione in Java trovata in rete [2].

Tale algoritmo consiste nella suddivisione della sequenza di partenza in sotto-sequenze (Runs) che vengono ordinate, se necessario, tramite l'algoritmo Insertionsort. Successivamente tali sequenze ordinate sono fuse in un'unica sequenza ordinata tramite l'algoritmo Mergesort.

Per prima cosa si provvede a calcolare la quantità `minrun`, che rappresenta la lunghezza minima di una Run, utilizzando il numero n di elementi. In particolare, la funzione `compute_minrun(n)` fa sì che la Run non sia nè troppo lunga (per rendere l'Insertionsort più efficiente), nè troppo corta (altrimenti si avrebbero troppe Merge da effettuare); inoltre è necessario che il numero di Runs sia una potenza di 2 (o un numero prossimo ad una potenza di 2), in modo da ottenere Runs di dimesioni simili per aumentare l'efficienza della fase di

Merge.

La fase successiva comporta la scansione della sequenza, cercando la sotto-sequenza più lunga possibile contenente elementi in ordine non decrescente o strettamente decrescente (in tal caso è sufficiente invertire la sequenza scambiando gli elementi ai due estremi).

Se la Run ottenuta ha una dimensione minore della quantità `minrun`, gli elementi mancanti vengono selezionati dagli elementi che seguono e la Run complessiva è ordinata tramite un Insertionsort binario, in cui la ricerca della posizione in cui inserire gli elementi avviene con una ricerca binaria. Dato che tale porzione è piccola e in parte già ordinata, l'Insertionsort è molto veloce.

La fase di Merge avviene sempre tra due Runs consecutive, in modo da garantire la stabilità dell'algoritmo. In particolare, appena una Run è stata identificata, viene aggiunto un elemento relativo a tale Run in uno stack, memorizzando l'indice di inizio della Run e la sua dimensione; successivamente viene richiamata la funzione `mergeCollapse`, che fa sì che le 3 Runs nella cima dello stack verifichino sempre le seguenti condizioni ($|X|$ indica il numero di elementi nella Run X):

- $|Z| > |Y| + |X|$
- $|Y| > |X|$

Se una delle condizioni non è rispettata, la Run Y è fusa con la Run più piccola tra X e Z. Questi accorgimenti sono mirati a far sì che le Runs abbiano tutte una dimensione simile.

La Merge tra due Runs consecutive è effettuata dalla funzione `mergeAt`.
In particolare, si identifica la Run più breve e si copia tale Run in un array temporaneo; successivamente si collocano opportunamente gli elementi nella sequenza d'uscita come in una Merge classica (da sinistra a destra o da destra a sinistra, a seconda se la prima Run è minore della seconda o viceversa).
La particolarità di tale algoritmo sta nella modifica della Merge, includendo la possibilità di effettuare una Merge con Galloping mode: si inizia la Merge con la modalità classica, ma si tiene traccia del numero di "vittorie" consecutive di una stessa Run (ovvero del numero di elementi consecutivi forniti da una stessa Run); se tale numero supera il valore prestabilito `minGallop` (7 di default), si può assumere che l'elemento successivo proverrà dalla stessa Run e si "entra" nel Galloping mode, ovvero si effettua una ricerca binaria sulla Run candidata a fornire l'elemento successivo dell'elemento dell'altra Run, in modo da poter spostare nella sequenza d'uscita porzioni più grandi delle Runs, invece di spostare un elemento alla volta ed effettuando meno confronti rispetto alla ricerca lineare. In tal caso si decrementa il valore `minGallop`, in modo che successivamente sarà più facile entrare nel Galloping mode.
Si resta nel Galloping mode fino a che una delle Run smette di vincere sempre rispetto all'altra. In tal caso si torna alla modalità classica e si incrementa il valore `minGallop`, in modo che successivamente sarà più difficile entrare nel Galloping mode.

Quando è stata scandita tutta la sequenza viene richiamata la funzione `mergeForceCollapse` che fonde le ultime Runs ordinate ancora separate.

Tale algoritmo ha una complessità pari a $\mathcal{O}(n \log n)$ nel caso peggiore e nel caso medio (come Mergesort), ma presenta una complessità pari a $\mathcal{O}(n)$ nel caso migliore (in cui gli elementi sono già ordinati).
In particolare, tale algoritmo risulta particolarmente efficiente se i dati da ordinare presentano un ordinamento pre-esistente e, dato che in numerosi contesti pratici i dati da ordinare presentano un ordine parziale, l'algoritmo Timsort può essere particolarmente efficiente nelle situazioni reali. [1]

L'algoritmo non è in loco, in quanto nella fase di Merge è utilizzata una memoria temporanea con dimensione pari alla Run più piccola.

L'implementazione realizzata è stabile, in quanto non vengono mai scambiati due elementi di uguale valore.

Viene riportato lo pseudo-codice dei passaggi salienti dell'algoritmo:

Algorithm 1 Timsort

```
1: function TIMSORT(A)                                   ▷ A è l'array da ordinare
2:     Stack S
3:     minrun = compute_minrun(n)
4:     baseAddress = 0
5:     for i = 0 to n do
6:         R = sotto-array di A più lungo possibile con elementi ordinati
7:         size = R.length
8:         if size < minrun then
9:             insertion_sort(R,A)                       ▷ InserionSort binario
10:            S.push(R)
11:            i = baseAddress + R.length
12:            baseAddress = i
13:        else
14:            S.push(R)
15:            i = i + 1
16:            baseAddress = i
17:        end if
18:        mergeCollapse(S,A)
19:    end for
20:    mergeForceCollapse(S,A)
21: end function
```

Algorithm 2 mergeCollapse

```
function mergeCollapse(S,A)    ▷ S è lo stack con le Runs, A è l'array da ordinare
    while S.length > 1 do
        n = S.length - 2
        if n > 0 & S[n − 1].length ≤ S[n].length + S[n + 1].length then
            if S[n − 1].length < S[n + 1].length then
                n = n − 1
            end if
            mergeAt(n, S, A)
        else if S[n].length ≤ S[n + 1].length then
            mergeAt(n, S, A)
        else
            break ▷ Si mantengono sempre le proprietà sui 3 elementi in cima allo stack
        end if
    end while
end function
```

Algorithm 3 mergeForceCollapse

```
function mergeForceCollapse(S,A)    ▷ S è lo stack con le Runs, A è l'array da ordinare
    while S.length > 1 do
        n = S.length - 2
        if n > 0 & S[n − 1].length < S[n + 1].length then
            n = n − 1
        end if
        mergeAt(n, S, A)
    end while
end function
```

Algorithm 4 mergeAt

function MERGEAT(n,S,A) ▷ n è l'elemento da fondere sullo stack con l'elemento n+1, S è lo stack con le Runs, A è l'array da ordinare
    $l1 = S[n]$.length
    $l2 = S[n+1]$.length
    $b1 = S[n]$.baseAddress
    $b2 = S[n+1]$.baseAddress
    Aggiorna S dopo la fusione

    Trova la poszione k in cui inserire $A[b2]$ all'interno della Run 1 ▷ Gallop
    $b1 = b1 + k$
    $l1 = l1 - k$
    Trova la poszione k in cui inserire $A[b1 + l1 - 1]$ all'interno della Run 2 ▷ Gallop
    $l2 = k$
    merge($A, b1, l1, b2, l2$)
end function

3 Verifica della correttezza delle implementazioni

Prima di pianificare gli esperimenti da condurre è stato creato uno script per verificare che tutte le implementazioni realizzate fossero corrette.

Per eseguire lo script (Verification.py) è necessario fornire da riga di comando tre parametri posizionali, nel seguente modo:

```
python3 ./Verification.py "file1.csv" "file2.csv" 0
```

oppure

```
python3 ./Verification.py "file1.csv" "file2.csv" 1
```

In particolare:

- file1.csv è il file csv contenente l'array disordinato
- file2.csv è il file csv contenente l'array ordinato, tramite una delle implementazioni, corrispondente a quello contenuto in file1.csv
- l'ultimo parametro va impostato a 0 se gli array contengono numeri, a 1 se contengono stringhe

I file csv contengono tutti gli elementi dell'array, separati da virgole.

Al fine di verificare la correttezza, nello script viene ordinato l'array contenuto in file1.csv tramite la funzione `sort()` di Python e viene confrontato il risultato con l'array contenuto in file2.csv, ordinato tramite una delle implementazioni realizzate.

4 Pianificazione dei test

4.1 Performance metric e Performance Indicator

La metrica di performance considerata è il **tempo**, mentre gli indicatori di performance asscociati ad essa sono stati il **numero di operazioni dominanti** eseguite, misurate incrementando un contatore, (livello astratto e indipendente dalla macchina che esegue l'algoritmo) e il **tempo di CPU** (livello concreto).

Dato che tutti gli algoritmi considerati sono basati su confronti, l'operazione dominante è l'operazione di confronto tra due elementi.

4.2 Parametri e Fattori

La misurazione del tempo impiegato dall'algoritmo può essere influenzata da numerosi parametri. Tutti gli esperimenti sono stati condotti con le seguenti configurazioni:

- **Hardware**:
  - CPU: Intel Core i5 dual-core a 2,3GHz
  - RAM: 8 GB 2133 MHz LPDDR3
- **Sistema operativo**: macOS Catalina, versione 10.15.4
- **Interprete Python**: versione 3.6

Per quanto riguarda la scelta dei fattori (parametri controllabili) sono state effettuate le seguenti scelte:

- **Taglia dell'input (n)**: per poter analizzare un trend nell'andamento del tempo d'esecuzione sono state considerate sequenze da ordinare di diverse dimensioni
- **Tipologia di input (type)**: per poter osservare le differenze di comportamento dei tre algoritmi sono stati selezionati inputs di diversa tipologia
- **minGallop**: per quanto riguarda l'algortimo Timsort è possibile considerare diversi valori della quantità minGallop, ovvero del numero di "vittorie" consecutive di una stessa Run prima di entrare nel Galloping mode

4.3 Livelli

I diversi valori assegnati ad ogni fattore sono i seguenti:

- **n**: come taglie sono state considerate potenze di 2, a partire dalla taglia $2^7 = 128$, fino alla taglia $2^{24} = 16777216$, ovvero n = (128, 256, 512, 1024, 2048,..., 16777216).
- **type**: le tipologie di input considerate sono le seguenti:
  - *int*: interi randomici (distribuzione uniforme discreta) nell'intervallo $[0, 2000]$

- *ord_asc*: reali nell'intervallo $[-2000, 2000)$ ordinati in ordine crescente
- *ord_desc*: reali nell'intervallo $[-2000, 2000)$ ordinati in ordine decrescente
- *perc*: reali nell'intervallo $[-2000, 2000)$, ordinati in ordine crescente, con sostituzione dell'1% delle entries con reali randomici (distribuzione uniforme nell'intervallo $[-2000, 2000)$). La scelta delle entries da sostituire avviene randomicamente
- *rand_normal*: reali randomici, con distribuzione normale standard
- *rand_uniform*: reali randomici, con distribuzione uniforme nell'intervallo $[-2000, 2000)$
- *str*: stringhe randomiche, ognuna con una lunghezza variabile (scelta randomicamente) da 3 a 6 caratteri [1]
- *window*: reali randomici, con distribuzione uniforme nell'intervallo $[-2000, 2000)$, con finestre pre-ordinate. La finestra ha una dimensione pari a $\frac{1}{10}$ della dimensione dell'array e le finestre pre-ordinate sono la prima, la quarta, la settima e la decima (immaginando di suddividere l'array in 10 finestre)

Di conseguenza: *type* = (int, ord_asc, ord_desc, perc, rand_normal, rand_uniform, str, window).

- **minGallop**: sebbene tale quantità sia impostata a 7 di default, in una fase della sperimentazione sono stati considerati i seguenti livelli: *minGallop* = (5, 100, 1000, 10000).

4.4 Generazione degli inputs

Le istanze di input da utilizzare per i test sono state generate tramite lo script input_gen.py, (nella cartella "inputs"), in accordo con i livelli sopra menzionati. Inoltre tali istanze sono state salvate, in formato csv, nella stessa cartella.
Ogni file csv contiene una diversa istanza di input, in cui i vari elementi sono separati da virgole.

5 Esecuzione dei test

5.1 Test sul tipo di dato

I primi test sono stati eseguiti per verificare che il comportamento delle tre implementazioni fosse indipendente dal tipo di dato da ordinare.

A tal fine sono stati considerati, per il fattore n, livelli da $2^7 = 128$ a $2^{22} = 4194304$ e, per il fattore *type*, i livelli int e str, ottenendo $16 \times 2 = 32$ design points per ognuna delle implementazioni (per il Timsort è stato mantenuto il fattore *minGallop* fisso a 7).

Tramite tali test sono quindi calcolate il numero di operazioni dominanti eseguite dai tre algoritmi [2], salvando i risultati dei test in un file csv (i file in questione si trovano nella cartella "instr_count", salvati con il formato nomeAlgoritmo_int.csv o nomeAlgoritmo_str.csv).

[1] in questo caso l'ordinamento dell'array finale è quello lessicografico

[2] la modalità d'esecuzione è specificata con il parametro `-e`, in questo caso fornendo il valore 2. Per le altre modalità d'esecuzione e parametri fare riferimento al file README.md

Successivamente sono stati plottati, tramite lo script Plot.py [3], i risultati ottenuti, in riferimento agli andamenti teorici previsti dagli algoritmi.

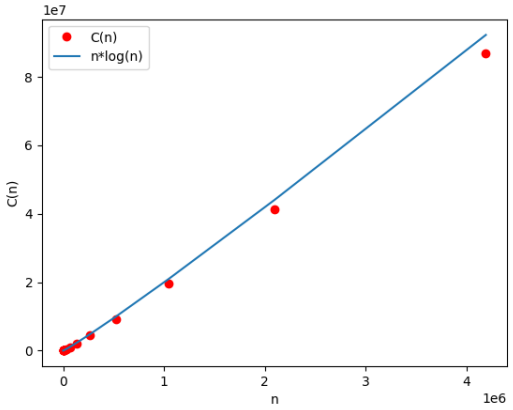

Figure 1: Mergesort con interi

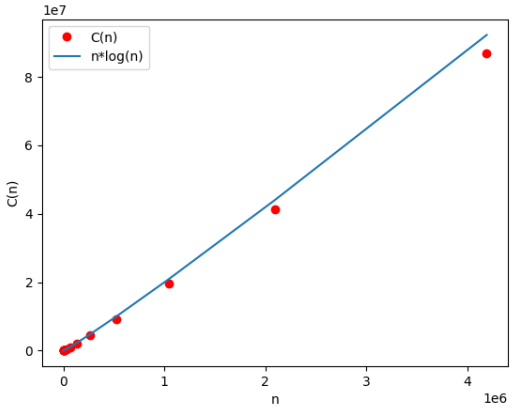

Figure 2: Mergesort con stringhe

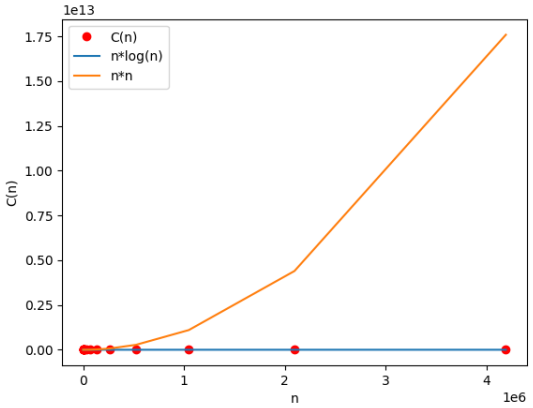

Figure 3: Quicksort con interi

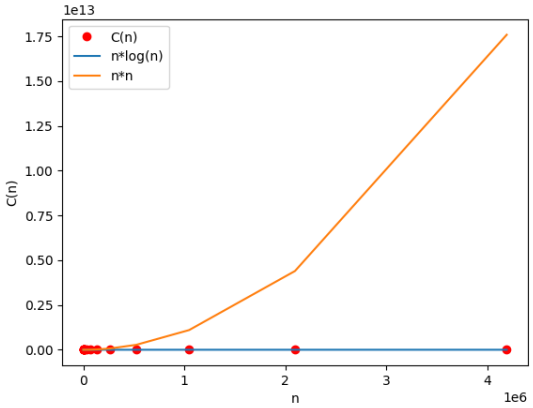

Figure 4: Quicksort con stringhe

[3] fare riferimento al file README.md per i dettagli sull'esecuzione

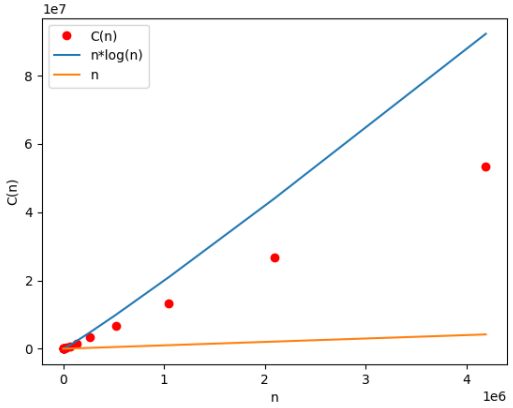

Figure 5: Timsort con interi

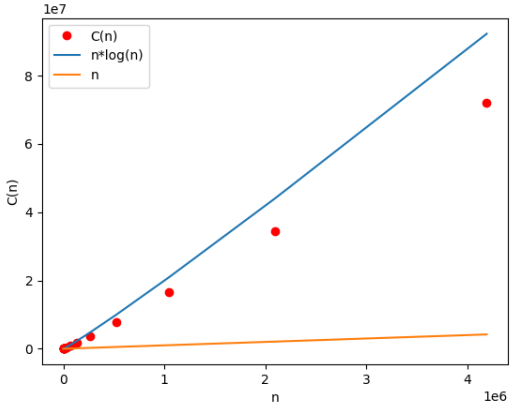

Figure 6: Timsort con stringhe

Considerazioni Quello che ho potuto osservare tramite questo test preliminare è che in tutti e tre i casi il numero di confronti effettuati rispecchia l'andamento teorico, sia per gli interi che per le stringhe.

In particolare per il Mergesort la quantità $C(n)$ segue l'andamento $\mathcal{O}(n \log n)$ (Figure 1 e 2); per il Quicksort vale un discorso analogo, infatti $C(n)$ è molto più vicino all'andamento $\mathcal{O}(n \log n)$ rispetto a $\mathcal{O}(n^2)$ (caso peggiore con array già ordinato, ma in questo caso l'input è scelto randomicamente) (Figure 3 e 4); per quanto riguarda il Timsort l'andamento di $C(n)$ è, in entrambi i casi, compreso fra $\mathcal{O}(n)$ (caso migliore) e $\mathcal{O}(n \log n)$ (caso peggiore e medio) (Figure 5 e 6).

5.2 Test sul numero di istruzioni dominanti

Dopo aver effettuato i test sopra-menzionati incentrati su due tipi di dato diversi tra loro, sono state prese in considerazione tutte le altre tipologie di input (numeri reali) e sono stati eseguiti test mirati al conteggio del numero di istruzioni dominanti eseguite.

In questa fase sono stati considerati tutti i livelli per quanto riguarda il fattore n, i livelli (ord_asc, ord_desc, perc, rand_normal, rand_uniform, window) per quanto riguarda il fattore *type* ed è stato mantenuto il fattore *minGallop* fisso a 7 per il Timsort.
Di conseguenza si otterrebbero $18 \times 6 = 108$ design points per ognuna delle implementazioni. Tuttavia in alcuni test non è stato possibile considerare tutti i livelli del fattore n a causa del tempo d'esecuzione troppo elevato; ad esempio, nell'esecuzione del Quicksort, per i livelli ord_asc e ord_desc del fattore *type*, è stato possibile considerare solo i primi 8 livelli del fattore n, ovvero $n = (128, 256, 512, ..., 16384)$, in quanto in questi due casi si ottiene il caso peggiore per il Quicksort, in cui la complessità è $\mathcal{O}(n^2)$.

I risultati dei test condotti sono anche in questo caso stati salvati in un file csv (cartella "instr_count", con il formato nomeAlgoritmo_type.csv).

Per prima cosa è stato utilizzato lo script MultiplePlots.py [4] per osservare il comportamento di ognuno degli algoritmi per le varie tipologie di input, ovvero per i vari livelli del fattore *type* considerati, ottenendo i seguenti risultati:

[4] fare riferimento al file README.md per i dettagli sull'esecuzione

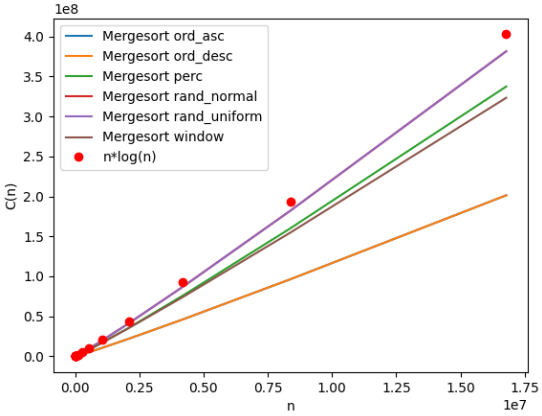

Figure 7: Mergesort

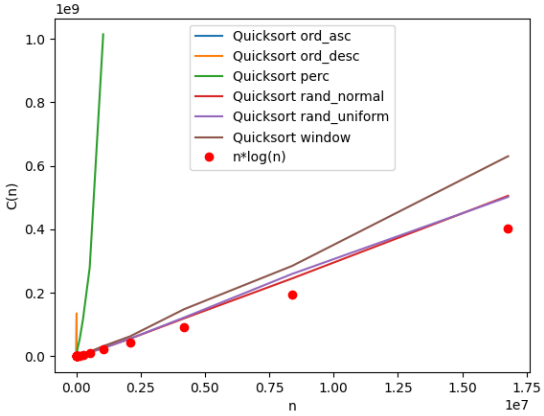

Figure 8: Quicksort

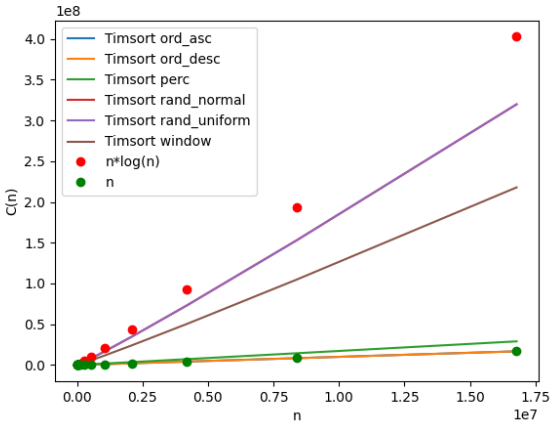

Figure 9: Timsort

Successivamente, sempre tramite lo script MultiplePlots.py, è stato effettuato il plot relativo ad ogni livello del fattore *type*, in modo da confrontare il comportamento dei tre algoritmi per una stessa tipologia di input, ottenendo i seguenti risultati:

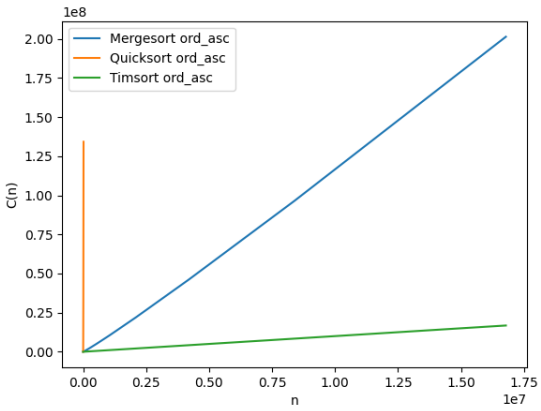

Figure 10: ord_asc

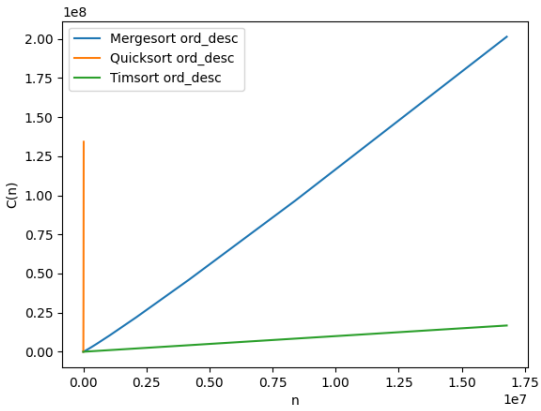

Figure 11: ord_desc

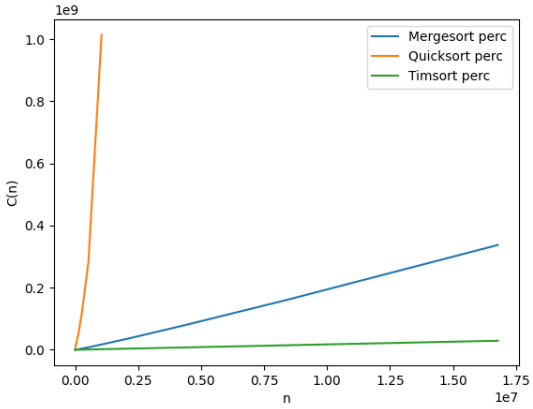

Figure 12: perc

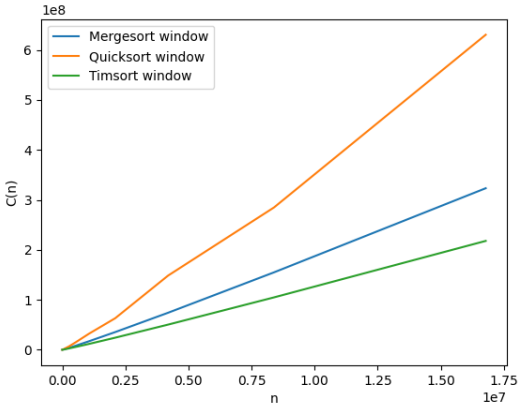

Figure 13: window

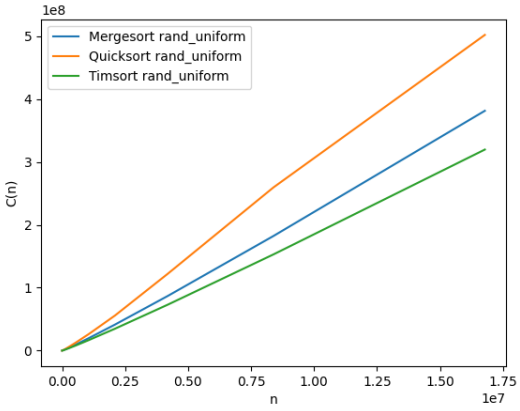

Figure 14: rand_uniform

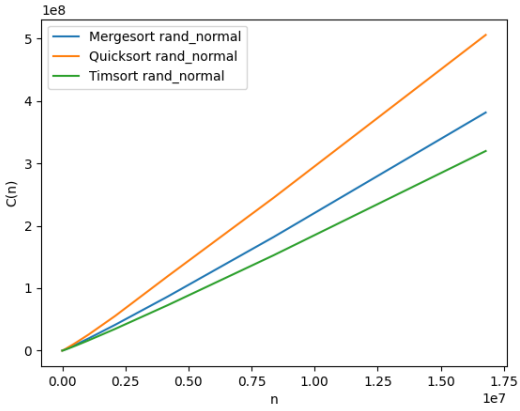

Figure 15: rand_normal

Considerazioni Per quanto riguarda il Mergesort (Figura 7) ho osservato un numero simile di confronti per i livelli rand_normal e rand_uniform, quindi nel caso in cui i dati siano

completamente randomici; inoltre il numero di confronti è lo stesso per i livelli ord_asc e ord_desc (caso migliore), ed è esattamente $\frac{1}{2}n \log n$, in quanto nella funzione `Merge` vengono selezionati sempre tutti gli elementi provenienti da uno dei sotto-array e gli elementi dell'altro sono inseriti in quello d'uscita senza effettuare confronti.

Nel Quicksort (Figura 8) il numero di confronti per i livelli ord_asc e ord_desc è quadratico, infatti produce il caso peggiore per l'algoritmo; con il livello perc viene comunque effettuato un numero elevato di confronti, in quanto si hanno array ordinati con alcune entries randomiche, quindi accade ancora spesso che l'elemento di "pivot" sia il più piccolo dell'intera sequenza (producendo partizioni sbilanciate); con dati randomici (livelli rand_normal e rand_uniform) l'algoritmo risulta particolarmente perfomante, avvicinandosi a un numero di confronti pari a $n \log n$ (caso medio).

Nel Timsort (Figura 9), per i livelli ord_asc e ord_desc si ottiene un numero di confronti lineare (caso migliore), dato che non si creano Runs da fondere e la sequenza è scandita tutta linearmente; con i livelli perc e window l'algoritmo può sfruttare un ordinamento preesistente dei dati; con i livelli rand_normal e rand_uniform il numero di confronti si avvicina a $n \log n$ e non c'è alcuna struttura nei dati che l'algoritmo può sfruttare.

5.3 Test sul tempo d'esecuzione

I test successivi che sono stati effettuati si collocano a un livello più basso della scala d'istanziazione, infatti sono stati eseguiti dei test mirati alla misurazione del tempo d'esecuzione (CPU time) dei vari algoritmi.

I livelli considerati in questa fase sono gli stessi utilizzati nei test precedenti sul numero di istruzioni dominanti eseguite; quindi, per ognuna delle implementazioni, si hanno gli stessi design points.

Per la misurazione del tempo di CPU è stata utilizzata la funzione `process_time()` del modulo `time`, campionando il tempo subito prima e subito dopo l'esecuzione dell'algoritmo e calcolandone la differenza.
Nei test precedenti il conteggio del numero di istruzioni dominanti non presenta variazione effettuando test ripetuti (***precisone*** massima, ma mancanza di ***accuratezza***); in questi test viene effettuata una misurazione molto più accurata, ma test ripetuti possono portare a misurazioni molto differenti tra loro, in quanto il tempo di CPU è influenzato da molti parametri (***accuratezza***, ma mancanza di ***precisione***). Per questo motivo, per il calcolo del tempo di CPU, vengono effettuate 3 misurazioni per ogni design point e ne viene calcolata la media.

I risultati dei test condotti sono stati salvati in formato csv (cartella "time_proc", con il formato nomeAlgoritmo_type.csv).

Analogamente al test precendete è stato prima effettuato un plot, tramite lo script MultiplePlots.py, per osservare il comportamento di ognuno degli algoritmi per le varie tipologie di input, ovvero per i vari livelli del fattore type considerati, ottenendo i seguenti risultati:

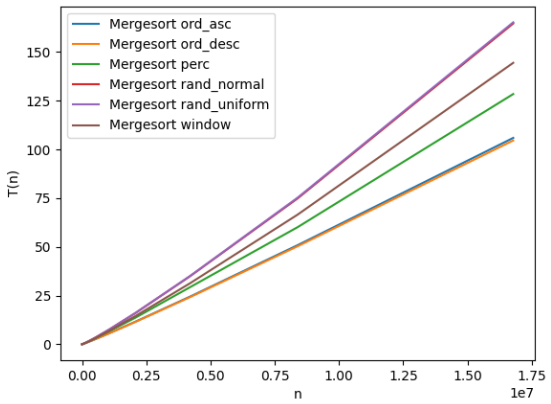

Figure 16: Mergesort

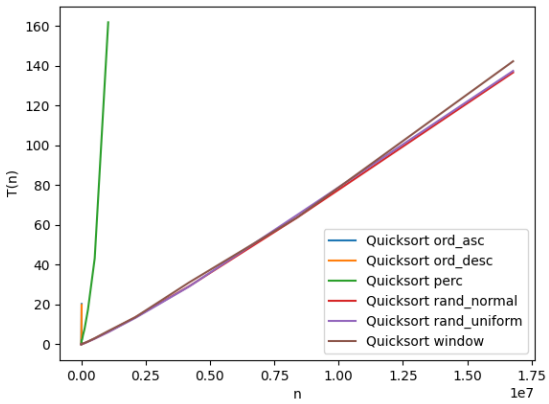

Figure 17: Quicksort

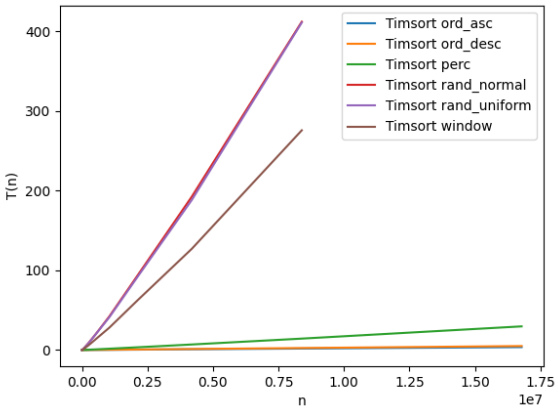

Figure 18: Timsort

Successivamente, sempre tramite lo script MultiplePlots.py, è stato effettuato il plot relativo ad ogni livello del fattore type, in modo da confrontare il comportamento dei tre algoritmi per una stessa tipologia di input, ottenendo i seguenti risultati:

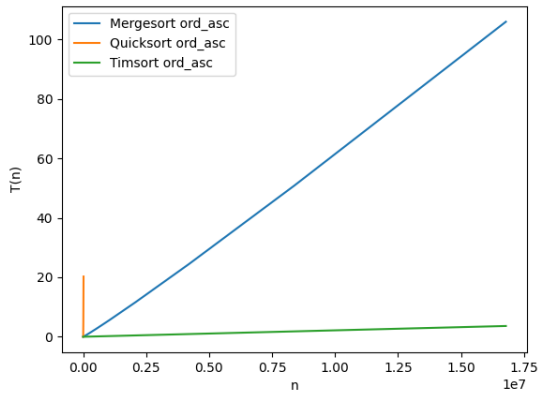

Figure 19: ord_asc

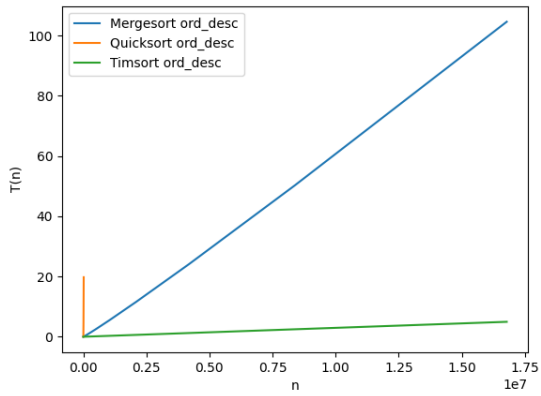

Figure 20: ord_desc

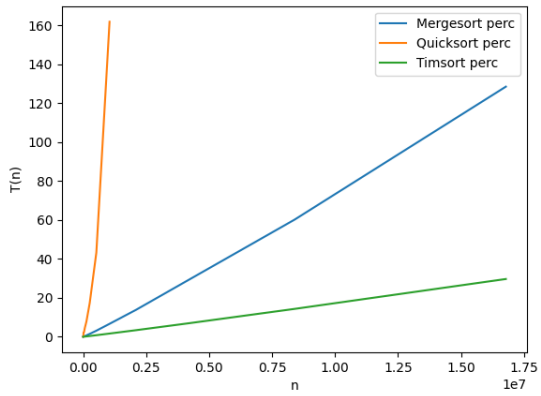

Figure 21: perc

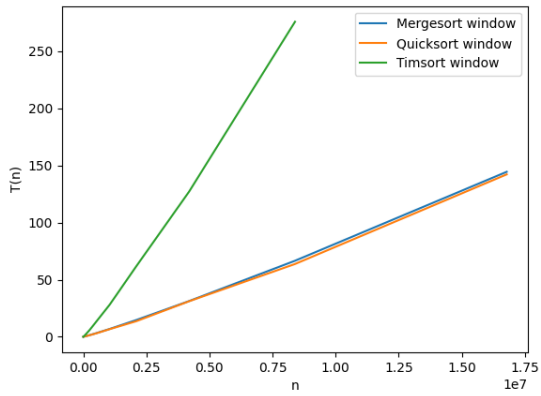

Figure 22: window

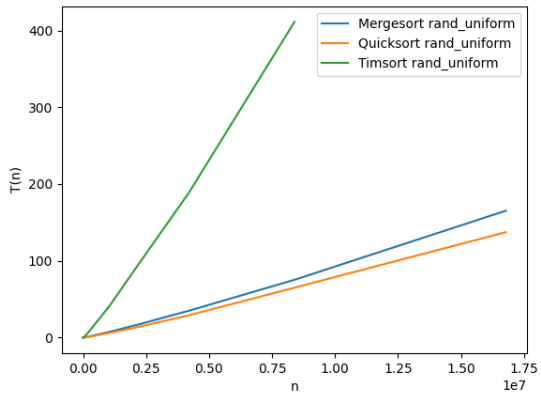

Figure 23: rand_uniform

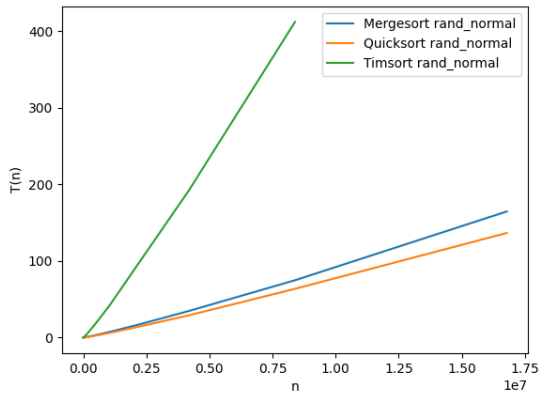

Figure 24: rand_normal

Considerazioni La prima cosa che ho potuto osservare confrontando le figure 17 e 18 con le figure 8 e 9 è che l'andamento del tempo di CPU per gli algoritmi Quicksort e Timsort

rispecchia l'andamento del numero di operazioni elementari eseguite per i vari livelli del fattore *type*. Tuttavia è da osservare che, nel caso del Timsort, sebbene per i tipi ord_asc e ord_desc il numero di confronti sia lo stesso, nel caso del tipo ord_desc il tempo impiegato è leggermente superiore, in quanto è necessario invertire la sequenza (funzione `reverse`).

Per quanto riguarda il Mergesort, confrontando la figura 16 e la figura 7 ho notato che, sebbene per il tipo window vengano effettuate meno operazioni di confronto rispetto al tipo perc, il tempo d'esecuzione calcolato è superiore nel tipo window rispetto al tipo perc.
Per analizzare in maniera più dettagliata questo aspetto ho utilizzato il profiler integrato nell'IDE (cProfile) per confrontare il tempo speso nelle varie funzioni nei due casi (la profilazione è stata eseguita per n = 16777216). I risultati ottenuti sono i seguenti:

Name	Call Count	Time (ms)	Own Time (ms)
Merge	16777215	119285 72,2%	119285 72,2%
<method 'writerow' of '_csv.writer' obj	1	20606 12,5%	20606 12,5%
MergeSort	33554431	133500 80,8%	14215 8,6%
MergeSort.py	1	165314 100,0%	10977 6,6%

Figure 25: cProfile per il tipo perc (Mergesort)

Name	Call Count	Time (ms)	Own Time (ms)
Merge	16777215	134937 72,9%	134937 72,9%
<method 'writerow' of '_csv.writer' obj	1	25449 13,8%	25449 13,8%
MergeSort	33554431	149730 80,9%	14792 8,0%
MergeSort.py	1	185054 100,0%	9682 5,2%

Figure 26: cProfile per il tipo window (Mergesort)

Quello che si può osservare tramite le figure 25 e 26 è che per il tipo window viene speso più tempo nella funzione `Merge` rispetto al tipo perc, a conferma dei tempi misurati tramite il test eseguito.
Da queste osservazioni deduco che ci sono aspetti che intervengono in fase d'esecuzione e che sono indipendenti dal numero di confronti che l'algoritmo esegue.

Infine, confrontando le figure 19, 20 e 21 rispettivamente con le figure 10, 11 e 12 ho potuto osservare che, per quanto riguarda i tipi ord_asc, ord_desc e perc, l'andamento del tempo di CPU per i tre algoritmi segue l'andamento trovato contando il numero di operazioni dominanti eseguite.

Confrontando le figure 22, 23 e 24 rispettivamente con le figure 13, 14 e 15 ho notato che per i tipi window, rand_uniform e rand_normal il tempo di CPU calcolato è discordante rispetto al numero di operazioni di confronto eseguite, soprattuto per quanto riguarda il Timsort: il test sul numero di operazioni dominanti mostra che in tutti e tre i casi l'algoritmo Timsort è quello che esegue meno operazioni di confronto, mentre il test sul tempo di CPU mostra che tale algoritmo impiega più tempo degli altri due.

Anche in questo caso ho utilizzato il profiler per capire in quali parti dell'esecuzione è speso più tempo; in particolare ho eseguito la profilazione dell'esecuzione del Timsort per n = 16777216 e per i tipi window e rand_uniform, ottenendo i seguenti risultati:

Name	Call Count	Time (ms)	Own Time (ms)
mergeLo	210030	384486 58,0%	192547 29,0%
gallopLeft	53673458	141593 21,4%	141593 21,4%
gallopRight	53830605	139603 21,1%	139603 21,1%
mergeHi	104548	203916 30,8%	101656 15,3%
<method 'writerow' of '_csv.writer' obj	1	24471 3,7%	24471 3,7%
binary_search	45369399	17656 2,7%	17656 2,7%
insertion_sort	314575	32416 4,9%	14760 2,2%
TimSort.py	1	663096 100,0%	10069 1,5%
<built-in method builtins.__build_clas	629156	9663 1,5%	9434 1,4%
TimSort	1	628310 94,8%	2576 0,4%
escapable	629156	12071 1,8%	2407 0,4%
mergeAt	314578	591914 89,3%	2029 0,3%
mergeCollapse	314579	591248 89,2%	861 0,1%

Figure 27: cProfile per il tipo window (Timsort)

Name	Call Count	Time (ms)	Own Time (ms)
mergeLo	350505	516018 53,6%	256258 26,6%
gallopRight	79686614	206662 21,5%	206662 21,5%
gallopLeft	79424278	205463 21,3%	205463 21,3%
mergeHi	173782	351323 36,5%	177239 18,4%
binary_search	75614005	28910 3,0%	28910 3,0%
<method 'writerow' of '_csv.writer' obj	1	23800 2,5%	23800 2,5%
insertion_sort	524288	51635 5,4%	22725 2,4%
<built-in method builtins.__build_clas	1048574	16000 1,7%	15637 1,6%
TimSort.py	1	962520 100,0%	9915 1,0%
escapable	1048574	20229 2,1%	4229 0,4%
mergeAt	524287	873065 90,7%	3300 0,3%
TimSort	1	928594 96,5%	1582 0,2%
mergeCollapse	524288	874703 90,9%	1389 0,1%

Figure 28: cProfile per il tipo rand_uniform (Timsort)

Come previsto i tempi d'esecuzione per il tipo window sono inferiori rispetto al tipo rand_uniform (come mostra la figura 18) ma, in entrambi i casi, le funzioni in cui viene speso più tempo sono: `mergeLo`, `mergeHi`, `gallopLeft` e `gallopRight`.
Alla luce di questi dati deduco che nell'esecuzione di tali funzioni non è sufficiente tenere conto di quanti confronti vengono eseguiti (incrementando il contatore), in quanto, come si può osservare dal codice, sono eseguite una serie di operazioni aggiuntive che sono state implementate da me seguendo la descrizione dell'algoritmo [1] e senza attuare particolari tecniche di ottimizzazione. Inoltre in fase d'esecuzione possono intervenire numerosi altri fattori, come ad esempio gli accessi alla memoria, che impattano in maniera considerevole sul tempo d'esecuzione complessivo dell'algoritmo.

5.4 Test sul fattore minGallop

L'ultimo test eseguito è stato incentrato sull'influenza che il fattore *minGallop* ha sul numero di operazioni di confronto eseguite dall'algoritmo Timsort.

In questa fase sono stati considerati i livelli da $2^{18} = 262144$ a $2^{22} = 4194204$ per il fattore n, ovvero n = (262144, 524288,...,4194204), i livelli (ord_asc, ord_desc, perc, rand_normal, rand_uniform, window) per quanto riguarda il fattore *type* e i livelli (5, 100, 1000, 10000) per il fattore *minGallop*.
Di conseguenza si hanno $5 \times 6 \times 4 = 120$ design points.

Tramite tali test sono state calcolate il numero di operazioni dominanti eseguite dal Timsort [5] per ognuno dei livelli scelti del fattore *minGallop*.

I risultati di tali test sono stati salvati in un file csv (cartella "minGallop/count").

Successivamente è stato utilizzato lo script minGallopPlot.py per visualizzare graficamente, al variare del fattore *minGallop*, il numero di confronti eseguiti dal Timsort, fissando i vari valori del fattore n, ottenendo i seguenti risultati:

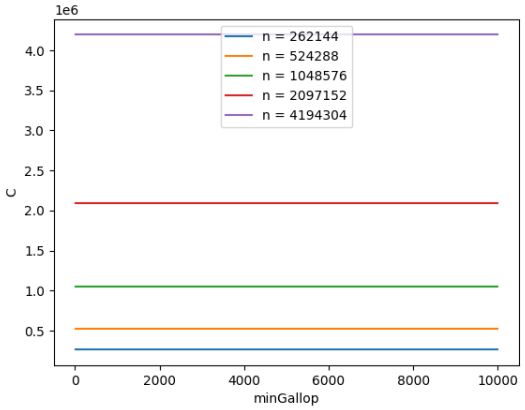

Figure 29: ord_asc

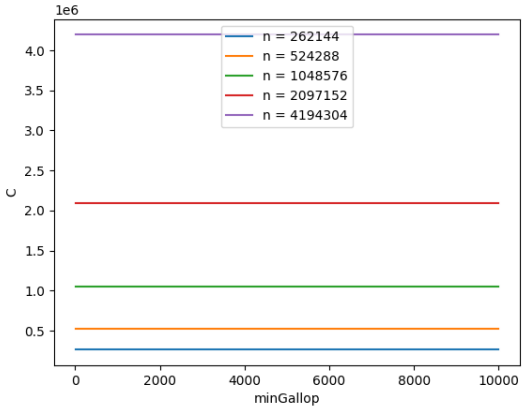

Figure 30: ord_desc

[5] la modalità d'esecuzione è specificata con il parametro `-e`, in questo caso fornendo il valore 4. Per le altre modalità d'esecuzione e parametri fare riferimento al file README.md

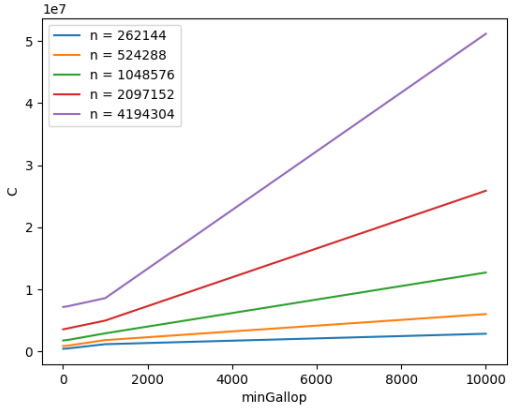

Figure 31: perc

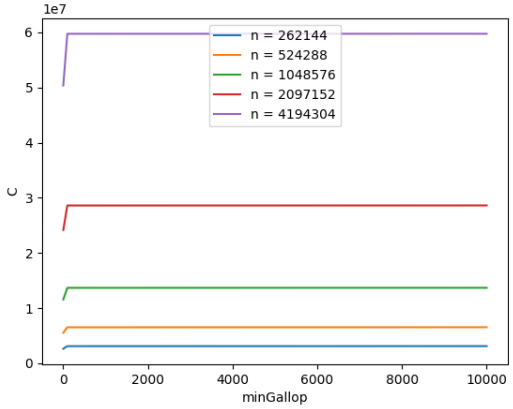

Figure 32: window

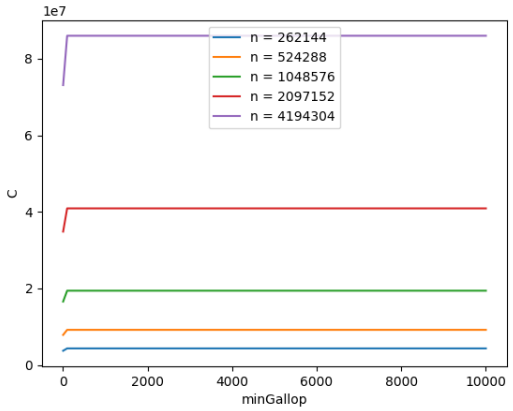

Figure 33: rand_uniform

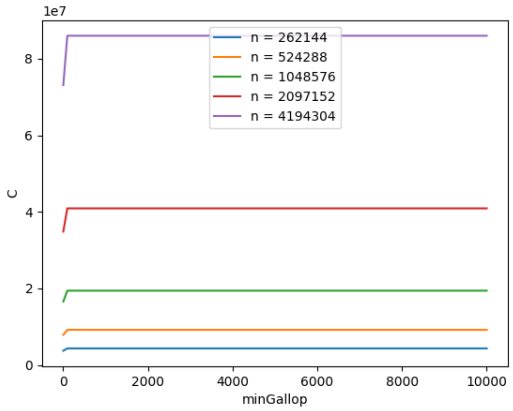

Figure 34: rand_normal

Considerazioni Come previsto per i tipi ord_asc e ord_desc il numero di confronti effettuati è costante al variare della quantità *minGallop*: in questi due casi viene scandita linearmente tutta la sequenza ottenendo un'unica Run, quindi non c'è bisogno di effettuare alcuna fusione e non si entra mai nel Galloping mode; per questo motivo la quantità *minGallop* non influenza il numero di confronti effettuati, che è sempre lineare con n (Figure 29 e 30).

Per quanto riguarda il tipo perc i dati hanno un ordinamento pre-esistente, quindi il Galloping mode può risultare conveniente per spostare, nella fase di fusione, porzioni più grandi della sequenza; per questo motivo, quando la quantità *minGallop* è più piccola, diminuisce il numero di "vittorie" consecutive che sono necessarie da parte di una stessa Run affinchè si entri nel Galloping mode e, per questo motivo, il numero di confronti aumenta all'aumentare di *minGallop* (Figura 31).

Per il tipo window si osserva che, all'aumentare di *minGallop*, aumenta il numero di confronti, anche se questo incremento è evidente nel passaggio di *minGallop* da 5 a 100 e quasi nullo per i restanti valori (Figura 32).

Infine per i tipi rand_normal e rand_uniform si osserva un incremento nel numero di confronti nel passaggio da 5 a 100, mentre per tutti gli altri valori di *minGallop* il numero

di confronti rimane costante; da ciò deduco che probabilmente in questi casi in cui i dati sono completamente randomici, quando *minGallop* = (100, 1000, 10000) non si entra mai nel Galloping mode, ovvero non accade mai che una stessa Run "vinca" per più di 100 volte. Come controllo aggiuntivo è stato verificato [6] quale fosse il valore di *minGallop* al termine di ogni esecuzione dell'algoritmo (i risultati sono stati salvati nella cartella "minGallop/m") e, per i tipi rand_normal e rand_uniform si ottengono i seguenti valori:

n	**minGallop = 5**	**minGallop = 100**	**minGallop = 1000**	**minGallop = 10000**
262144	1	100	1000	10000
524288	1	100	1000	10000
1048576	1	100	1000	10000
2097152	1	100	1000	10000
4194304	1	100	1000	10000

I dati riportati confermano che la quantità *minGallop* al termine dell'esecuzione non cambia rispetto al valore assegnato per *minGallop* = (100, 1000, 10000), quindi si può affermare che probabilmente non si entra mai nel Galloping mode per questi valori di *minGallop*.

6 Conclusioni e sviluppi futuri

Riassumendo, il lavoro che ho svolto è iniziato con uno studio del funzionamento degli algoritmi Mergesort, Quicksort e Timsort e un'analisi della complessità teorica degli stessi.

Successivamente ho provveduto ad implementare in Python gli algoritmi sopra-menzionati e a verificarne la correttezza.

Nella fase successiva ho pianificato quali test condurre, identificando la metrica di performance e i corrispondenti indicatori di performance, oltre che i parametri, i fattori e i rispettivi livelli su cui effettuare i test. Inoltre ho provveduto a generare gli inputs necessari.

Nella fase di test ho dapprima effettuato delle verifiche mirate all'analisi del comportamento degli algoritmi con tipi diversi di dati di input (interi e stringhe) e successivamente ho effettuato dei test mirati al conteggio del numero di istruzioni dominanti (confronti) eseguite dagli algoritmi (verificando se le aspettative teoriche fossero soddisfatte); successivamente ho condotto dei test mirati alla misurazione del tempo di CPU delle implementazioni realizzate e ho confrontato i risultati ottenuti con quelli dei test precedenti. Infine ho eseguito dei test facendo variare il fattore *minGallop* dell'algoritmo Timsort ed ho analizzato come il numero di confronti eseguiti è influenzato da tale fattore.

Lo studio condotto ha permesso di evidenziare come l'approccio puramente teoretico alla progettazione e analisi degli algoritmi presenta delle limitazioni dovute alle assunzioni che vengono fatte in fase di analisi.

Alla luce di queste considerazioni ritengo che potrebbe essere interessante considerare i seguenti sviluppi futuri:

[6] la modalità d'esecuzione è specificata con il parametro `-e`, in questo caso fornendo il valore 5. Per le altre modalità d'esecuzione e parametri fare riferimento al file README.md

- Analizzare in maniera più approfondita i casi in cui la misura del tempo di CPU si è discostata dagli andamenti predetti tramite il conteggio delle istruzioni dominanti
- Individuare costanti e considerare termini di ordine inferiore nell'espressione della complessità per avere una stima più precisa del comportamento degli algoritmi in fase d'esecuzione
- Estendere lo studio considerando un numero maggiore di livelli per il fattore n e per il fattore *type* (considerando quindi anche altre tipologie di input)

References

[1] Descrizione del Timsort: https://github.com/python/cpython/blob/master/Objects/listsort.txt.

[2] Implementazione del Timsort in java: https://www.codota.com/web/assistant/code/rs/5c76a236e70f87