

UNIVERSITÀ DEGLI STUDI DI VERONA

---

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI  
Corso di Laurea in Informatica

Tesi di Laurea

**ALGORITMI DI ORDINAMENTO:  
ANALISI E RIDUZIONE DEL  
CONSUMO ENERGETICO NEI  
DISPOSITIVI MOBILI**



Relatore:

Chiar.mo Prof. **Romeo Rizzi**

Correlatore:

Chiar.mo Prof. **Ferdinando Cicalese**

Laureando:

**Matteo Olivato**

Anno Accademico **2014\2015**



## Sommario

L'indipendenza energetica dei dispositivi mobili sta diventando un argomento prevalente per l'avanzamento del mercato tecnologico mondiale. Molti ricercatori si stanno concentrando sulla fisica dei dispositivi integrati e sulla chimica delle batterie. Poco si sta ricercando in ambito software, anche se è la componente che rende effettivamente utile e produttivo un qualsiasi dispositivo embedded. L'obiettivo è di presentare una dimostrazione empirica dei miglioramenti energetici, relativi ad un attento studio del software utilizzato, attraverso algoritmi semplici ma esemplificativi quali quelli di ordinamento. L'utilizzo di Raspberry Pi 2 Model B ci ha permesso di effettuare misurazioni su un'architettura analoga a quelle tuttora in vigore per la maggior parte dei dispositivi mobili. I risultati dei test evidenziano chiaramente come la scelta del software e una sua ottimizzazione per ridurre i consumi energetici, porti all'allungamento del tempo di vita delle batterie. Ogni altro ambito informatico potrà avvalersi di questo studio per considerare utili, se non addirittura essenziali, attività di ricerca energetica sul software e su tutti algoritmi in generale.



# Indice

<b>1</b>	<b>Dispositivo utilizzato e ambiente di test</b>	<b>1</b>
1.1	Introduzione . . . . .	1
1.2	Dispositivo utilizzato . . . . .	2
1.3	Algoritmi scelti . . . . .	3
1.4	L'ambiente di test . . . . .	4
1.5	Rappresentazione e comparazione dei risultati . . . . .	6
<b>2</b>	<b>Aree di ricerca</b>	<b>7</b>
2.1	Considerazioni utili e ambiti di ricerca . . . . .	7
<b>3</b>	<b>Standard swap e Xor swap</b>	<b>13</b>
3.1	Panoramica generale . . . . .	13
3.2	Implementazioni utilizzate . . . . .	13
3.3	Risultati e grafici . . . . .	14
<b>4</b>	<b>Minimizzare le scritture</b>	<b>17</b>
4.1	Panoramica generale . . . . .	17
4.2	Il CycleSort . . . . .	18
4.2.1	Struttura e idea dell'algoritmo . . . . .	18
4.2.2	Complessità e numero totale di scritture . . . . .	18
4.3	L' OlivatoCycleSort . . . . .	20
4.3.1	Struttura e idea dell'algoritmo . . . . .	20
4.3.2	Complessità e numero totale di scritture . . . . .	22
4.4	Test energetici e risultati . . . . .	24
4.5	Interesse del caso di studio . . . . .	25
<b>5</b>	<b>Algoritmi non uniformi</b>	<b>27</b>
5.1	Panoramica generale . . . . .	27
5.2	Algoritmo di Ford-Johnson (Merge-Insert) . . . . .	27
5.2.1	Considerazioni generali . . . . .	28
5.3	Algoritmo non uniforme ideale . . . . .	28

5.4	Innesto in algoritmi uniformi . . . . .	29
5.4.1	Algoritmo candidato all'integrazione . . . . .	30
5.5	Implementazioni integrate . . . . .	31
5.5.1	Funzione non uniforme avanzata . . . . .	37
5.5.1.1	Idea dell'algoritmo avanzato . . . . .	37
5.5.1.2	Descrizione dettagliata . . . . .	38
5.5.1.3	Implementazione e integrazione dell'algoritmo . . . . .	39
5.6	Test e Grafici . . . . .	42
<b>6</b>	<b>Conclusioni</b>	<b>47</b>
6.1	Conclusione e prospettive future . . . . .	47
	<b>Riferimenti bibliografici</b>	<b>50</b>

# Elenco delle figure

1.1	Raspberry Pi 2 Model B . . . . .	3
3.1	Scambi effettuati prima dello scaricamento completo della batteria di riferimento. Le tre diverse configurazioni rappresentano il numero totale di scambi effettuati in una esecuzione. . . . .	15
3.2	Durata della batteria di riferimento nelle tre diverse configurazioni. . . . .	15
4.1	Numero totale di elementi ordinati prima del completo scaricamento della batteria di riferimento utilizzando tre differenti lunghezze di vettore. . . . .	25
4.2	Durata della batteria di riferimento rispetto alle tre lunghezze di vettore. . . . .	25
5.1	Numero totale di elementi ordinati prima del completo scaricamento della batteria di riferimento rispetto alla funzione qsort dell' ANSI C. . . . .	43
5.2	Durata della batteria di riferimento rispetto alla funzione qsort dell' ANSI C. . . . .	43
5.3	Numero totale di elementi ordinati prima del completo scaricamento della batteria di riferimento rispetto al quicksort. . . . .	44
5.4	Differenze nella durata della batteria di riferimento rispetto al quicksort. . . . .	44





# Capitolo 1

## Dispositivo utilizzato e ambiente di test

### 1.1 Introduzione

Negli ultimi anni abbiamo visto un forte incremento dei dispositivi embedded e mobile che hanno rivoluzionato l'interazione tra l'uomo e l'informatica. Questi dispositivi diventano ogni giorno sempre più potenti e raffinati, tanto da poter supportare la gestione di un vasto numero di sensori mediante sistemi operativi sempre più complessi. La rapidità nell'avanzamento tecnologico non è stata omogenea in tutte le sue componenti. Un esempio lampante può essere il ritmo di miglioramento nelle nuove versioni del software e delle nuove funzionalità hardware, che non sono seguite da un adeguato miglioramento in termini di efficienza energetica. Siamo passati nel giro di pochi anni da dispositivi mobili con processori monocore, 512 megabyte di RAM con fotocamera da 2 Mpixel a dispositivi con processori octa-core, 4GB di RAM e fotocamere da 21 Mpixel. Un incremento notevole nell'ambito delle funzionalità non accompagnato da un proporzionale aumento dell'autonomia energetica. E' semplice capire come inizi ad essere necessario un incremento del tempo di vita di tali dispositivi, rendendoli indipendenti, per tempi più lunghi possibili, da una fonte di energia diretta. Soluzioni in questo ambito sono ricercate da tempo nei campi della fisica, della chimica delle batterie e dei dispositivi integrati. Si ricercano soluzioni che incrementino la densità energetica e processi produttivi che riducano l'area dei transistor per diminuire i consumi e poter aumentare la frequenza di lavoro senza eccessive ripercussioni sull'affidabilità.

Grazie ai miglioramenti hardware, iniziano ad affacciarsi sul mercato dispositivi (smartwach, Google Glasses, Microsoft Hololens, ecc...) con un'intrinseca natura mobile, che aspirano ad una indipendenza energetica

sempre più duratura. Il raggiungimento di un'efficienza adeguata, allo stato attuale, non è ancora stato raggiunto e i limiti hardware col passare degli anni si fanno sempre più stringenti e costosi.

Un'idea potrebbe essere il cambiamento del punto di vista e ambito di ricerca, esaminando l'altra faccia della medaglia: il Software. Le domande che sorgono spontanee sono:

Perché non cercare di capire come i diversi algoritmi impattano sulle prestazioni e durata delle batterie?

Perché non cercare modifiche ad algoritmi esistenti per aumentare l'efficienza energetica senza dover preoccuparci dell'hardware?

Lo scopo di questo testo è di provare a dare una risposta, anche se limitata e parziale, a queste domande, proponendo soluzioni che possano essere utili in questi contesti.

## 1.2 Dispositivo utilizzato

Per metterci nelle condizioni di poter effettuare modifiche sul software in maniera significativa, dobbiamo prima scegliere un dispositivo hardware con le seguenti caratteristiche:

1. Deve contenere un processore rappresentativo di quelli utilizzati in ambito mobile.
2. Deve poter permettere una misurazione del consumo rapida ed affidabile, limitando al minimo componenti hardware parassiti che andrebbe ad influire sulle misurazioni.
3. Deve poter essere configurabile a livello software, permettendo così una eliminazione dei processi inutilizzati.

Dopo un'analisi attenta e dettagliata si è scelta la Raspberry Pi 2 model B[15]. Qui sotto sono descritte le caratteristiche tecniche:

- Processore: 900MHz quad-core ARM Cortex-A7 CPU (ARMv7)
- Memoria centrale : 1GB LPDDR2 SDRAM
- Dimensioni della scheda: 85.60 mm × 56.5 mm
- Memoria di massa: microSD
- Consumo massimo stimato: 800 mA (4 Watt)

Come possiamo notare il dispositivo in questione è dotato di un processore ARM quad-core, in linea con i più recenti dispositivi embedded e mobile e di 1 GB di SDRAM, una quantità analoga a quella installata sulla maggior parte di dispositivi presenti sul mercato attuale. Le misure contenute e compatte permettono una facile manipolazione, una minore dispersione energetica da parte dei componenti, nonché una più facile schermatura da eventuali influenze dall'ambiente esterno (campi magnetici).

Nel lato software troviamo Il sistema operativo Raspbian[16] una distribuzione Linux basata sul famoso Debian. Questa versione di sistema operativo, essendo basata su Linux, ha la flessibilità per poter essere modificata e personalizzata per ridurre il quantitativo di software “inutile”, che con processi in background potrebbe falsare l’effettiva oggettività della misurazione.

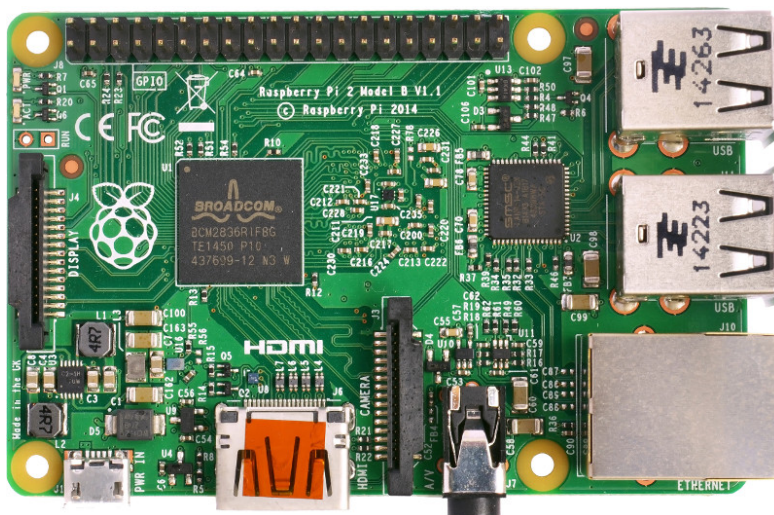


Figura 1.1: Raspberry Pi 2 Model B

### 1.3 Algoritmi scelti

Per rendere i test significativi anche per gli aspetti algoritmici, dopo un'analisi attenta, si è deciso di andare ad indagare il consumo e suo possibile miglioramento nell'ambito degli Algoritmi di Ordinamento. Questo per diversi motivi:

- Sono algoritmi semplici da implementare e modificare.
- L'ambiente di test e di debug di una modifica risulta estremamente affidabile.

## 4 CAPITOLO 1. DISPOSITIVO UTILIZZATO E AMBIENTE DI TEST

- Il codice degli algoritmi è generalmente contenuto, quindi facilmente caricabile nella sua interezza nella memoria centrale.
- Gli algoritmi di ordinamento sono utilizzati in molti altri algoritmi più complessi, a partire da algoritmi di ricerca, fino ad arrivare ad algoritmi anche molto raffinati sui grafi, rendendo quindi una loro analisi ed eventuale modifica vista sotto una prospettiva energetica direttamente vantaggiosa in molti altri contesti.

Tra tutti i possibili algoritmi di ordinamento si sono voluti analizzare solo quelli che avessero avuto un'importanza e un'efficacia in una loro possibile modifica utile e significativa, eliminando quegli algoritmi che per natura avrebbero fornito dei dati poco significativi. Di seguito si presenta l'elenco degli algoritmi e delle considerazioni atte a far comprendere la scelta di inserirli nel progetto di test:

### **Introsort[6] e Quicksort[13]**

Il primo algoritmo è quello utilizzato di default nella libreria standard del C, che quindi risulta essere il punto di riferimento da considerare, ed eventualmente da sfidare, nell'ambito di un miglioramento dei consumi. Il secondo è la base algoritmica del primo e sarà quindi utilizzato come base per delle eventuali modifiche.

### **CycleSort[3, 5] e OlivatoCycleSort**

Questi algoritmi hanno come idea di base la minimizzazione del numero di scritture durante il processo di ordinamento. Più avanti vedremo che senso ha tenerli in considerazione e in quali ottiche e contesti potrebbero risultare interessanti.

## 1.4 L'ambiente di test

Le caratteristiche dell'ambiente di test sono state scelte con alcune considerazioni descritte in seguito dettagliatamente.

Ambiente Software:

### 1. **Sistema Operativo Raspbian:**

Sistema operativo derivato da Debian e personalizzato rimuovendo i pacchetti inutili e limitando l'avvio di processi e servizi che avrebbero potuto inquinare la veridicità delle misure.

### 2. **Servizio SSH:**

Questo metodo di connessione remota si è preferito alla connessione

diretta (monitor, mouse, tastiera ecc..) a causa di una significativa riduzione dei consumi standard del dispositivo. L'unico elemento che potrebbe introdurre consumo è il dongle wifi, ma nelle nostre considerazioni la differenza di consumo tra quest'ultimo e la somma degli altri dispositivi necessari per un controllo diretto, era nettamente a favore di quest'ultimo. Inoltre l'interfaccia a riga di comando ci ha permesso di non dover attivare il processo XWindows normalmente adibito alla composizione della GUI, così da ridurre ulteriormente i processi in background non di interesse per i nostri test.

### 3. *Compilatore GCC:*

Gli algoritmi e le loro versioni sono state tutte scritte in linguaggio C, essendo il linguaggio più utilizzato per la scrittura dei Kernel dei Sistemi Operativi e quindi il più significativo in vista di modifiche e ottimizzazioni di carattere energetico che, se inserite direttamente nel Kernel del sistema, potrebbero effettivamente portare a qualche miglioramento nell'ambito dei consumi tutti gli applicativi installati. La compilazione utilizza il flag -O3, che rappresenta il massimo livello di ottimizzazione applicabile dal compilatore sul programma. Esisterebbero altri flag specifici per ottimizzazioni su processori ARM ma si è scelto di non utilizzarli perché renderebbero troppo specifico il codice eseguibile generato.

Ambiente Hardware:

#### 1. *Raspberry Pi 2 model B:*

Questo dispositivo già stato descritto in precedenza.

#### 2. *USB Wifi Dongle:*

Utilizzato per consentire la connessione remota via SSH.

#### 3. *Alimentatore 5V-1A e cavo USB:*

Necessari all'alimentazione del dispositivo.

#### 4. *Prolunga a doppia presa:*

Una prolunga modificata avente una sola spina e due prese, la prima utilizzata per la connessione dell'amperometro di precisione, la seconda utilizzata per il collegamento dell'alimentatore di Raspberry Pi 2.

#### 5. *Amperometro:*

Risulta indispensabile un amperometro di precisione, con misure sulla corrente alternata fino a 0.01mA, che sarà collegato in serie al nostro dispositivo.

## 1.5 Rappresentazione e comparazione dei risultati

I risultati dei test hanno due scopi significativi. Il primo verificare il consumo medio e il secondo l'efficienza media del dispositivo durante l'esecuzione degli algoritmi. Un algoritmo che potrebbe avere un consumo medio molto basso, potrebbe non essere efficiente perché il numero totale di elementi ordinati potrebbe essere molto inferiore al numero di elementi ordinati da un algoritmo con consumo medio più alto. Queste due caratteristiche sono entrambe interessanti, perciò saranno trattate separatamente.

Per rendere il confronto sui consumi energetici più realistico e poter comprendere meglio i grafici abbiamo comparato il consumo al tempo di vita di una batteria. Per lo scopo si è considerata una batteria da 5V e 1000mAh. Grazie a questa considerazione possiamo conoscere facilmente la durata di una batteria sotto l'esecuzione di un determinato algoritmo e le differenze tra le varie modifiche. Per quanto riguarda l'efficienza, nei grafici sarà espressa come numero totale di elementi ordinati prima dell'esaurimento completo della batteria. Si cercherà inoltre di trarre delle considerazioni finali confrontando i due aspetti.

# Capitolo 2

## Aree di ricerca

### 2.1 Considerazioni utili e ambiti di ricerca

Per arrivare ad una diminuzione dei consumi nell'esecuzione di un algoritmo, per prima cosa dobbiamo valutare dei metodi di approccio che possano poi essere verificati scientificamente. L'idea di base consiste nell'osservare quali caratteristiche algoritmiche possano portare ad un consumo energetico inferiore a parità di tempo di esecuzione (più o meno piccole variazioni). Di seguito saranno descritte alcune caratteristiche di interesse:

1. *Numero totale di confronti:*

Il cuore di molti algoritmi e soprattutto di quelli di ordinamento sono i confronti. Per sua natura l'operazione di confronto avviene tra due elementi e deve generare un risultato booleano. I due elementi coinvolti nel confronto dovranno quindi essere letti dalla memoria, successivamente devono essere confrontati e il risultato del confronto deve essere scritto in un registro per poi poter essere effettivamente utilizzato. Possiamo perciò vedere il confronto come due letture, un confronto e una scrittura. Ognuna di queste tre operazioni evidentemente consuma energia e la loro somma è sicuramente la componente che influisce maggiormente sul risultato energetico totale dell'algoritmo.

Una strategia di interesse è quindi quella di ridurre la complessità dell'algoritmo, agendo principalmente sulla diminuzione de complessità nel caso pessimo ma anche quella nel caso migliore. Avremmo così sicuramente un miglioramento nel caso medio, compreso tra la massima e la minima complessità, che potrebbe dare risultati significativi per quanto riguarda l'ambito dei consumi.

2. *Numero totale di scritture:*

Una caratteristica poco ricercata ma che potrebbe essere cruciale dal

punto di vista dei consumi energetici è il numero di scritture effettuate durante l'esecuzione dell'algoritmo. Una scrittura in memoria, a causa della necessità di modificare lo stato di un gruppo di bit, dovrà utilizzare una tensione e corrente superiore in confronto a quelle utilizzate durante il processo di lettura. E' possibile dire che normalmente il valore energetico associato alle scritture è circa tre volte quello associato alle letture. Quindi un'ipotesi da cui possiamo partire è che il numero di scritture totali in memoria influisca direttamente sul consumo energetico di un algoritmo e che ridurre il numero di scritture totali eseguite, potrebbe essere una buona strategia per migliorare la sua efficienza energetica.

3. ***Dimensione dell'algoritmo:***

Un'altra caratteristica da tenere in considerazione è la dimensione dell'eseguibile compilato. I processori mobile, normalmente, sono dotati di una memoria cache ridotta, che, non riuscendo a contenere tutto il programma, potrebbe andare in contro al fenomeno dei cache-miss[1]. Ogni cache-miss, durante l'esecuzione del codice, provoca una trascrizione di blocchi dalla memoria centrale alla memoria Cache. L'energia è spesa nel leggere e scrivere aree di memoria che, se fossero contenute tutte all'interno della Cache, sarebbero lette e caricate una sola volta all'avvio del processo.

Un algoritmo di piccole dimensioni permette il suo caricamento completo all'interno della memoria Cache, risultando molto più rapido da un punto di vista temporale, ma anche molto più efficiente da quello energetico. Dunque una strategia efficiente potrebbe essere quella di monitorare la dimensione del codice affinché rimanga sotto i limiti dimensionali della cache.

4. ***Località dell'algoritmo:***

Sebbene le considerazioni precedenti siano già molto efficaci possiamo anche pensare ad un aspetto ulteriore. La Cache come tipo di memoria è organizzata su due o tre livelli. Nel livello più esterno ci sarà la maggior parte del codice da eseguire, con in aggiunta la parte dei dati, mentre nei livelli più interni ci saranno quei blocchi che conterranno il gruppo di dati e codice attualmente in esecuzione. Più precisamente nei livelli più interni avremo:

- (a) I blocchi di codice che conterranno un intorno del comando puntato dal Program-Counter, ovvero quello in esecuzione in quel momento.
- (b) Un intorno della posizione nella struttura dati a cui si sta effettuando l'accesso, sia che sia in lettura, sia che sia in scrittura.



Consideriamo quindi l'idea che un algoritmo che sfrutti il principio di località, sia per quanto riguarda il codice da eseguire, sia nell'accesso alla struttura dati utilizzata, più efficiente a livello energetico. Riducendo il numero di cache-miss tra i vari livelli interni alla cache si diminuisce il numero di letture e scritture, responsabili di una buona parte del consumo energetico.

5. *Memoria ausiliaria utilizzata:*

Gli algoritmi di ordinamento come molti altri algoritmi eseguono le loro operazioni su una struttura dati. Per portare a termine il processo possono avvalersi di memoria aggiuntiva a tempo di esecuzione, oppure lavorare sulla struttura dati senza utilizzare memoria ulteriore se non una quantità costante allocata all'avvio del processo stesso.

Come si potrebbe facilmente dedurre, l'operazione di aggiunta di ulteriore memoria a tempo di esecuzione, comporterà un ovvio utilizzo della stessa, che si trasformerà in un numero di letture e scritture arbitrario. Si deve inoltre sapere che l'allocazione di ulteriore memoria sullo stack è effettuata come richiesta al sistema operativo che a sua volta per decidere quale locazione fisica assegnare deve eseguire dei complessi algoritmi di gestione della memoria. E' ovvio che in queste condizioni l'algoritmo, oltre a subire dei rallentamenti per quanto riguarda il tempo di esecuzione, aumenta l'energia totale spesa riducendo significativamente la sua efficienza. Un' algoritmo che di natura non utilizzi memoria ausiliaria risulta preferibile e nettamente più efficiente per il nostro caso di studio.

6. *Complessità della struttura dati:*

Il componente su cui lavorano quasi tutti i tipi di algoritmi, specialmente quelli di ordinamento, consiste in una struttura dati. Non sempre ci si accorge che la complessità della struttura dati utilizzata può essere causa di inefficienza energetica. Basti pensare che una struttura dati complessa potrebbe richiedere un reindirizzamento multiplo prima di arrivare al dato contenuto al suo interno, che si vuole manipolare.

Se vista nell'ambito delle letture e scritture potremmo dire che il numero di letture e scritture del nuovo indirizzo di memoria da recuperare può essere arbitrario a seconda della profondità della struttura dati. Ci sono due ulteriori aspetti che di riflesso incidono sull'efficienza. Il primo riguarda la maggior dimensione del codice dell'algoritmo che deve tener conto della vera posizione, nella struttura dati, del dato a cui dover accedere e che quindi aumenterà la lunghezza e il numero dei suoi comandi a seconda delle regole sintattiche imposte dal linguaggio. La seconda si

riferisce alla dimensione in memoria di una struttura dati complessa, che può risultare fino a  $n$  volte maggiore di una struttura dati semplice. Quest'ultime due considerazioni se viste nell'ottica della dimensione dell'algoritmo e della località dell'algoritmo risultano evidentemente deleterie e controproducenti. Il numero di cache-miss aumenterà con l'aumentare della complessità della struttura dati sotto tutti e due i punti di vista.

La politica adottata sarà quindi quella di utilizzare strutture dati il più semplici possibili. Sarà perciò utilizzato l'array di Interi.

Alla luce di queste considerazioni si sono cercati dei campi applicativi che potessero migliorare uno o più dei precedenti ambiti. Il risultato è la ricerca su tre diversi aspetti implementativi e di modifica degli algoritmi:

1. *Standard swap[17] vs Xor swap[20]:*

All'interno degli algoritmi di ordinamento una cosa molto importante è sicuramente la funzione che effettua gli scambi. Il metodo generalmente utilizzato è quello di utilizzare una variabile temporanea di supporto che permetta la transizione di un valore per poi essere riscritto successivamente nella sua posizione finale. Questo metodo prevede l'uso di tre variabili, due che devono essere scambiate e una terza che fa da appoggio, che chiameremo Standard swap.

Un ipotetico miglioramento della funzione atta ad effettuare gli scambi risulterebbe notevolmente utile in quanto migliorerebbe per effetto tutti gli algoritmi di ordinamento che ne fanno uso.

Per il nostro caso si sono cercati dei metodi equivalenti che avrebbero permesso gli scambi ma che confrontati con lo Standard swap avrebbero portato a qualche tipo di miglioramento energetico, sia a livello di minor tempo di esecuzione che nell'efficienza energetica complessiva.

2. *Min comparison sort[7] e algoritmi di ordinamento non uniformi:*

Il problema dell'ordinamento per confronti a livello di complessità è un problema che risiede tra  $\Omega(N)$  e  $O(N \log_2 N)$ , e questo grazie al fatto che, in alcuni casi (es: vettore già ordinato), sfruttando le proprietà transitive della relazione di ordinamento si può concludere che la struttura dati sia ordinata con meno degli standard  $N \log_2 N$  confronti. Il problema del Minimum Comparison Sort corrisponde al trovare la funzione di ordinamento ottima, cioè con complessità minima, data una particolare lunghezza del vettore. Perciò considerando l'algoritmo ottimo in termini di confronti si possono ottimizzare il numero di scritture effettuate nella singola permutazione così da effettuare il minor numero di scritture

a fronte del minor numero di confronti. Questo approccio è chiamato non uniforme in quanto non si conosce l'esecuzione sequenziale che avrà l'algoritmo, ogni ramo può saltare a zone di codice diverse tra loro a seconda del risultato del confronto e quindi del tipo di permutazione analizzata. A causa della crescita esponenziale dell'albero dei confronti ottimo e la complessità nel generarlo per numeri al di sopra della decina, un algoritmo di questo genere non può essere utilizzato per ordinare ottimamente un numero arbitrario di elementi, avendo l'ulteriore problema di aumentare la dimensione del codice esponenzialmente all'aumentare del numero di elementi andando ad influire sulle proprietà di dimensione e quindi di caricamento e swap dalla memoria del codice da eseguire.

Così si è pensato ad una integrazione di funzioni che implementassero il Minimum Comparison Sort per piccole dimensioni all'interno di altri algoritmi di ordinamento, sfruttando come organizzazione generale il codice dell'algoritmo di ordinamento ospite ma ottimizzando l'ordinamento locale di una piccola porzione dell'array con la funzione di Minimum Comparison Sort.

A questo scopo l'algoritmo Quicksort è risultato un ottimo candidato ai test di questa soluzione in quanto già utilizzato come base nell'Introsort nostro punto di riferimento.

### 3. *Algoritmi con numero minimo di scritture:*

Come è possibile immaginare la problematica del numero di scritture risulta particolarmente affascinante, sia dal punto di vista energetico che per il miglioramento della velocità di esecuzione. Seppur gli algoritmi si preoccupino della loro complessità, dobbiamo tenere presente che la somma delle scritture risulta particolarmente importante a livello energetico, soprattutto perché quest'ultime potrebbero essere effettuate non solamente sulla memoria primaria ma anche su supporti di memoria secondaria, più costosi in termini di tempo e energia per l'accesso e la modifica.

Esistono inoltre campi interessanti di ricerca nella Teoria dell'Informazione[8] che ipotizzano, con opportune considerazioni teoriche e considerazioni su una possibile revisione delle nostre architetture, di poter annullare il costo delle letture in termini energetici e in alcuni casi anche temporali ( come la computazione quantistica e il fenomeno dell'entanglement quantistico ). Si denota come il costo energetico risulterebbe legato solamente all'atto delle scritture e che quindi una ricerca di algoritmi ottimi in tal senso avrebbe delle ripercussioni sulle prospettive di miglioramento future.



# Capitolo 3

## Standard swap e Xor swap

### 3.1 Panoramica generale

Il primo campo di ricerca su chi si è lavorato è il problema degli scambi. Questa problematica è primaria nell'esecuzione della maggior parte degli algoritmi di ordinamento. Sicuramente, essendo così basilare, una sua ottimizzazione otterrebbe risultati apprezzabili su tutti quegli algoritmi che ne fanno uso. La strategia di scambio più utilizzata è sicuramente quella che sfrutta una variabile temporanea. In alcuni casi, soprattutto quello numerico, esistono altri tipi di scambio che però sono legati a operazioni da effettuare tra le variabili da scambiare. Possiamo citarne un paio:

- Scambio tramite somma e sottrazione tra interi[18]
- Scambio tramite Xor[20]

Il nostro caso abbiamo scelto come struttura dati l' Array di Interi, il che ci permetterebbe di utilizzare entrambi i tipi per effettuare i test. Si è deciso di utilizzare come possibile candidato solamente il secondo in quanto il primo metodo è legato solamente a tipi di dato intero, mentre il secondo è applicabile a tutti i tipi di dato primitivi del C (char, integer, float, ecc..). Inoltre le operazioni di somma e sottrazioni tra interi sono sicuramente più complesse da effettuare che una operazione di Bitwise Xor richiesta dal secondo metodo.

### 3.2 Implementazioni utilizzate

Abbiamo scelto di utilizzare come implementazione di test un algoritmo molto semplice. All'interno di un ciclo andremo a scambiare tra loro due variabili utilizzando nel primo caso lo Standard swap[17] come funzione di scambio

e nel secondo caso utilizzando la funzione che chiameremo Xor swap. La seconda funzione effettuerà ripetutamente uno Xor bit a bit (Bitwise Xor) tra gli elementi da scambiare ottenendo come risultato lo scambio dei due valori. Come sappiamo il Bitwise Xor può essere effettuato su qualsiasi tipo di dato visto che non guarda alla natura del dato ma va a lavorare direttamente sulla sua codifica binaria e dunque sicuramente anche su tipi primitivi come gli interi. Il codice è il seguente:

```

1 int main(void) {
2
3     //Numero di iterazioni da eseguire
4     int NMAX = 1000000001;
5     //Variabili a e b da scambiare
6     int a=2, b=7, c, i;
7
8     for (i=0;i<NMAX;i++){
9         //Eseguo lo scambio utilizzando la variabile c di supporto
10        c = a;
11        a = b;
12        b = c;
13    }
14 }
```

Listing 3.1: Codice C dell'implementazione Standard Swap

```

1 int main(void) {
2
3     //Numero di iterazioni da eseguire
4     int NMAX = 1000000001;
5     int a=2, b=7, c, i;
6
7     for (i=0;i<NMAX;i++){
8         //Scambio le variabili utilizzando la proprietà dello Xor
9         a = a ^ b;
10        b = a ^ b;
11        a = a ^ b;
12    }
13 }
```

Listing 3.2: Codice C dell'implementazione Xor Swap

### 3.3 Risultati e grafici

I risultati di questi test sono abbastanza espliciti. Il metodo dello Standard swap vince su tutti i fronti. Non solo è estremamente più veloce ma ha anche un consumo nettamente inferiore del metodo Xor swap. I grafici sono esplicativi:

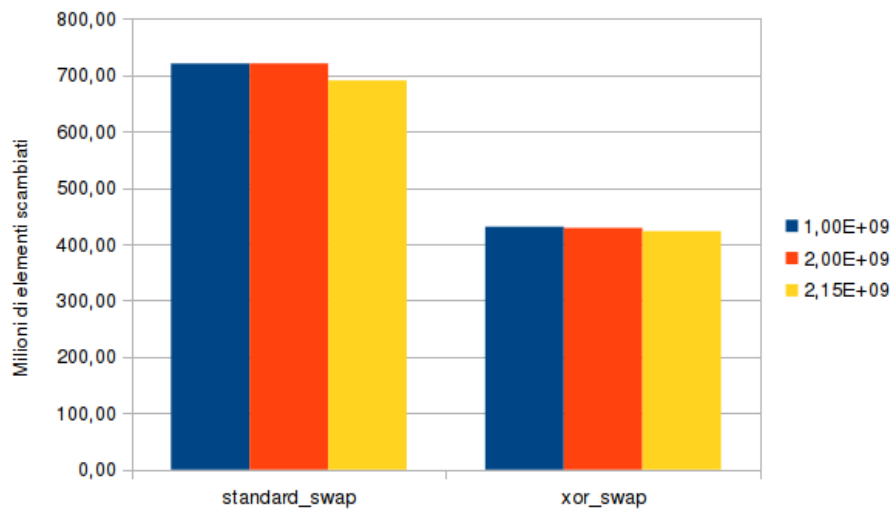


Figura 3.1: Scambi effettuati prima dello scaricamento completo della batteria di riferimento. Le tre diverse configurazioni rappresentano il numero totale di scambi effettuati in una esecuzione.

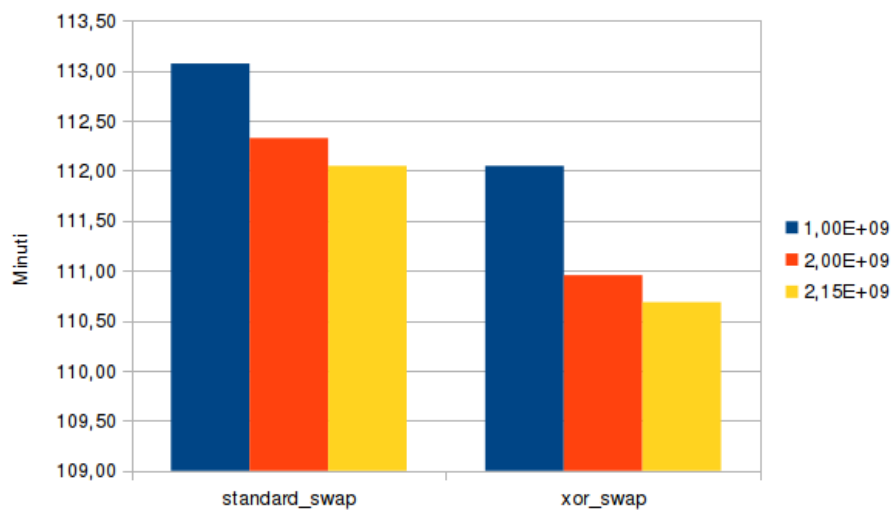


Figura 3.2: Durata della batteria di riferimento nelle tre diverse configurazioni.

Andando a studiare i risultati a posteriori possiamo suggerire delle motivazioni ragionevoli che confermano la veridicità del risultato. Più precisamente:

- Il metodo Standard swap può essere visto come la seguente somma:

$$1 \text{ allocazione di variabile intera} + 3 \text{ assegnamenti}$$

Andando ad analizzare le ottimizzazioni effettuabili dal compilatore si capisce come l'allocazione di una variabile intera ai fini di scambio può essere trasformata nell'utilizzo di un registro interno come supporto allo scambio, evitando l'allocazione e l'utilizzo di memoria sullo stack. I registri del processore sono gli elementi di memoria più veloci ed efficienti in assoluto su qualsiasi architettura. Perciò il consumo significativo avviene solamente nel momento degli assegnamenti.

- Il metodo Xor swap può essere visto come la seguente somma:

$$3 \text{ assegnamenti} + 3 \text{ operazioni di BitwiseXor tra interi}$$

Come possiamo notare oltre a degli assegnamenti abbiamo il carico computazionale di 3 operazioni di Bitwise Xor.

Andando a vedere quindi la differenza tra le due funzioni notiamo fondamentalmente un eccesso nella seconda funzione di 3 operazioni di Bitwise Xor non banali. E' quindi comprensibile come il secondo metodo riduca sia l'efficienza dal punto di vista energetico sia l'efficienza per quanto riguarda la velocità di esecuzione. Il metodo di Bitwise Xor potrebbe essere preferibile quando il processore è dotato di implementazioni molto efficienti dell'operazione di Bitwise Xor e non possa disporre di un registro di appoggio per uno scambio efficiente. A fronte della situazione reale si può tranquillamente affermare che il metodo Standard è il metodo più efficiente sotto tutti i punti di vista.



# Capitolo 4

## Minimizzare le scritture

### 4.1 Panoramica generale

Sulla problematica energetica abbiamo già accennato all'importanza del numero totale di scritture. Si sono cercati degli algoritmi candidati in questo senso. Sono stati scelti due algoritmi di ordinamento, uno dei quali completamente inedito, che promettono il minor numero di scritture in assoluto:

1. *CycleSort*[3, 5]

2. *OlivatoCycleSort*

Questi due algoritmi hanno in comune molte caratteristiche già note di interesse ovvero:

- Sono tutti e due algoritmi In-place ovvero che non usano memoria secondaria.
- Sono tutti e due algoritmi con un codice di esecuzione contenuto.
- Hanno una buona località a livello di codice di esecuzione e dei dati.

Sono tutti e due algoritmi che presentano una complessità quadratica, ovvero  $O(N^2)$ , con alcune differenze. Il primo supera decisamente la complessità quadratica arrivando più o meno al valore doppio mentre il secondo algoritmo, che è una variante ottimizzata del primo e prodotto dal relatore di questo testo, ha una complessità compresa tra  $\Omega(N)$  e  $O(N^2)$  esatti.

Queste differenze nella complessità incideranno moltissimo sui risultati sia temporali che energetici. Ricordiamoci che i confronti sono abbastanza onerosi sia energeticamente che da un punto di vista computazionale e che quindi un valore esponenziale non può che essere preponderante confrontato con la linearità delle scritture. Ma andiamo comunque a descriverli in dettaglio.

## 4.2 Il CycleSort

Il CycleSort[5] è un algoritmo relativamente recente e poco conosciuto. Fonda la sua idea di esecuzione esattamente sull'ottenere il minor numero di scritture totali. Riesce ad ottenere non solo il minor numero di scritture tra tutti gli algoritmi di ordinamento più conosciuti, ma addirittura ottiene, in generale, il minimo numero possibile. Se la lunghezza del vettore è di  $N$  elementi, il CycleSort effettuerà al massimo  $N$  scritture.

### 4.2.1 Struttura e idea dell'algoritmo

L'algoritmo basa la sua idea sui cicli di ordinamento. Partendo dalla destra del vettore, dato un elemento in posizione  $p$  nell'array, se questo non è ordinato vuol dire che la sua posizione finale sarà diversa da  $p$ . Riposizionando l'elemento nella sua posizione finale  $f$ , possiamo reiterare il ragionamento sull'elemento che occupava quella posizione, il quale ovviamente non potrà essere ordinato visto che occupava la posizione di un altro elemento. Continuando a reiterare arriveremo a trovare l'elemento non ordinato, che aveva proprio come sua posizione finale la posizione  $p$  di partenza del nostro ciclo. Una volta che siamo tornati alla posizione  $p$  di partenza il nostro ciclo si è chiuso e avremo ordinato un certo numero di elementi. Se mantenendo il conto degli elementi globalmente ordinati il valore è  $N$ , allora sappiamo che abbiamo terminato l'ordinamento, in quanto abbiamo riposizionato nella loro posizione finale tutti gli  $N$  elementi dell'array. Nel caso il contatore sia diverso da  $N$  allora all'interno del mio array esisterà un altro ciclo di ordinamento che non è ancora stato effettuato. Purtroppo non sapendo quali elementi successivi alla mia posizione di partenza  $p$  ho ordinato, devo ricontrollare iterativamente tutti gli elementi che incontro nel restante sottoarray a sinistra.

### 4.2.2 Complessità e numero totale di scritture

Per quanto riguarda la complessità abbiamo un valore più che quadratico. Infatti partendo dal primo elemento e dovendo confrontarlo con tutti gli altri per trovare la sua esatta posizione, abbiamo in tutto  $N$  confronti. A causa del fatto che nei cicli di ordinamento l'algoritmo non conosce gli elementi già ordinati deve ogni volta ricontrollarli tutti e  $N$ . Questo fa sì che non solo durante i cicli di ordinamento ma anche nei successivi cicli di ricontrollo l'algoritmo scorra completamente o parzialmente l'array. Da questo è facile dedurre come la complessità totale superi  $O(N^2)$  rimanendo comunque in questo ordine di grandezza. Nelle scritture il caso migliore si riscontra quando il vettore è già ordinato con un totale di 0 scritture. Il caso peggiore invece si

raggiunge con una particolare combinazione di elementi che banalmente può essere rappresentata dal caso esemplificativo del vettore ordinato a cui si è applicato uno shift a destra di tutti gli elementi. Il massimo che verrebbe escluso dall'array è collocato in prima posizione. In quest'ultimo caso otterremo N scritture.

```

1 //Funzione che ordina con il minor numero di scritture nel
  vettore
2 int cyclesort(int v[], int nmax){
3   int i, j, change, temp, temppos;
4
5   //Scorro gli elementi nel vettore
6   for(i=0; i<nmax; i++){
7     temp=v[i];
8
9     //Valuto la posizione in cui dovrò inserire l'elemento
10    temppos = i;
11    for(j=i+1; j<nmax; j++)
12      if(temp>v[j]) temppos++;
13
14    //Se l'elemento è già nella posizione ordinata non lo
      riscrivo e continuo
15    if(temppos == i) continue;
16
17    //Altrimenti inizio a riposizionare gli elementi nella loro
      posizione corretta
18    while(temppos != i){
19
20      //Se ho più elementi con lo stesso valore rimappo la
        posizione
21      while(temp == v[temppos]) temppos++;
22
23      //Inserisco l'elemento nella sua posizione
24      change = v[temppos];
25      v[temppos]=temp;
26      temp=change;
27
28      //Trovo la mia prossima posizione
29      temppos = i;
30      for(j=i+1; j<nmax; j++)
31        if(temp>v[j]) temppos++;
32    }
33
34    //Scrivo nella posizione di arrivo l'elemento salvato in temp
35    v[temppos]=temp;
36  }
37 }
38
39 //Creo un vettore riempito di numeri casuali

```

```

40 void generate_vector(int v[], int nmax, int intervallo){
41     int i;
42
43     //Inizializzo il randomizzatore
44     srand(time(NULL));
45
46     //Inizializzo il vettore con numeri casuali
47     for(i=0;i<nmax;i++)
48         v[i]=rand() % intervallo;
49 }
50
51 int main(int argc, char *argv[]) {
52
53     //Recupero il numero di elementi da generare
54     int nmax;
55     nmax = atoi(argv[1]);
56     int intervallo = nmax;
57     int v[nmax];
58
59     //Genero un vettore casuale di lunghezza nmax
60     generate_vector(v, nmax, intervallo);
61
62     //Eseguo l'algoritmo
63     cyclesort(v,nmax);
64 }

```

Listing 4.1: Codice C dell'implementazione del CycleSort

## 4.3 L' OlivatoCycleSort

Questo algoritmo, da me ideato, ha come idea di base i cicli di ordinamento come il CycleSort. Nonostante ciò approccia in modo nettamente differente al problema dell'ordinamento cercando di ridurre sia la complessità minima del CycleSort sia quella massima.

### 4.3.1 Struttura e idea dell'algoritmo

L'algoritmo si struttura in due passi principali. Il primo passo, consiste nello scorrere l'array partendo da destra fino a che gli elementi tra loro soddisfino la proprietà di ordinamento parziale, con cui si è scelto di ordinare. Se durante lo scorrimento troviamo un elemento che non soddisfa l'ordinamento parziale, abbiamo trovato un elemento minore del massimo elemento nel sottoarray di destra.

Questa condizione ci dice che esiste una elevata probabilità che l'elemento appena trovato non sia nella sua posizione corretta in riferimento all'ordi-

namento globale e una certezza sul fatto che tutti gli elementi maggiori dell'elemento trovato che erano contenuti nel sottoarray di destra non sono sicuramente ordinati rispetto all'ordine globale. Possiamo inoltre dedurre che se una parte degli elementi di destra non è sicuramente ordinata, vuol dire che esisterà nella parte di sinistra uno o più elementi non ordinati che saranno degli elementi di un ciclo di ordinamento non ancora effettuato che li andrà ad ordinare.

Il secondo passo, alla luce di queste considerazioni, consisterà nell'andare ad eseguire un ciclo di ordinamento scegliendo l'elemento appena trovato esterno al sottoarray di destra. Come risultato avremmo due casi:

I. Nel primo caso l'elemento ha trovato un ciclo di ordinamento e ha ordinato, compreso se stesso, un gruppo di elementi. Da qui dobbiamo considerare altri due casi:

- (1) Il mio ciclo più quelli passati hanno ordinato globalmente  $N$  elementi e quindi sono nel caso in cui il vettore è ordinato e termino.
- (2) Il mio ciclo più i precedenti ha ordinato un numero minore di  $N$  elementi e quindi potrebbero esistere ulteriori elementi da ordinare, ovvero potrebbe esistere un elemento appartenente ad un ciclo di ordinamento non ancora eseguito.

Dalle precedenti considerazioni posso affermare che se non ho ordinato tutti gli elementi del mio array con questo ciclo potrebbe esistere un elemento non ordinato globalmente. Distinguiamo anche qui due casi:

- (1) Se ripartendo con un ciclo che ricerca gli ordinamenti parziali, proprio dalla posizione appena ordinata globalmente, non troviamo nessun elemento non ordinato, questo implica che la porzione a sinistra dell'elemento è ordinata rispetto a quest'ultimo. Ma visto che siamo partiti da un elemento che è ordinato globalmente se tutti gli elementi successivi sono ordinati rispetto ad esso allora sono obbligatoriamente ordinati anche globalmente.
- (2) Se ripartendo con un ciclo che ricerca gli ordinamenti parziali proprio dalla posizione appena ordinata globalmente, troviamo un elemento non ordinato questo sarà un possibile candidato per iniziare un nuovo ciclo di ordinamento, tornando ricorsivamente alla dimostrazione di correttezza globale dell'algoritmo.

II. Nel secondo caso l'elemento non ha trovato un ciclo da eseguire. Sicuramente è già posizionato correttamente rispetto all'ordinamento globale. Il che vuol dire che oltre agli elementi minori di se stesso contenuti nel

sottoarray ordinato di destra esistono esattamente  $m$  elementi minori di se stesso a sinistra, con  $m$  il numero di elementi maggiori di se stesso contenuti nel sottoarray di destra. Alla luce di queste semplici deduzioni possiamo affermare che se facessimo ripartire il ciclo di ricerca di un ordinamento parziale proprio da questo elemento prima o poi ci imbatteremo in un elemento minore di se stesso che sappiamo essere sicuramente presente a sinistra. L'elemento nuovamente trovato è un nuovo possibile candidato a poter iniziare un nuovo ciclo di ordinamento, tornando ricorsivamente alla dimostrazione di correttezza globale dell'algoritmo.

### 4.3.2 Complessità e numero totale di scritture

La complessità di questo algoritmo è sicuramente nell'ordine di  $O(N^2)$ . Possiamo però essere più precisi dire che è compresa tra  $\Omega(N)$  e  $O(N^2)$ . Il caso in cui la complessità risulta minima è il caso banale in cui l'array sia già ordinato. In questo caso in fatti il ciclo di ordinamento parziale noterà che partendo dall'inizio dell'array è arrivato alla fine senza aver effettuato cicli di ordinamento, ma ha solamente confrontato transitivamente gli elementi.

Il caso peggiore si ottiene quando si considera un vettore ordinato e si applica uno shift a destra degli elementi di una posizione e ricollocando l'elemento massimo, che uscirebbe dall'array, in prima posizione. In questo caso il primo ciclo di ricerca di sottoarray parziali scoprirebbe che il secondo elemento è un possibile candidato per avviare un ciclo di ordinamento. Una volta avviato il ciclo riposizionerà a catena tutti gli elementi nella loro posizione globalmente corretta, per farlo per ogni elemento dovrà confrontarlo con tutti gli altri  $N - 1$  elementi dell'array. Una volta terminato questo ciclo il contatore noterà che ha ordinato esattamente  $N$  elementi e perciò terminerà.

Se riguardiamo, a parte il primo confronto che nota il secondo elemento come candidato per un possibile ciclo, poi al suo interno il ciclo che risulterà risolutivo per ogni elemento effettuerà  $N - 1$  confronti, il che risulta esattamente  $N * N$  ovvero  $N^2$ .

Quello che ad una analisi più approfondita si può notare è che il caso medio di complessità di questo algoritmo è decisamente inferiore al CycleSort classico, soprattutto in presenza di un array quasi ordinato.

Per quanto riguarda il numero di scritture possiamo dire che nel caso migliore non effettuerà nessuna scrittura, ovvero il caso in cui il vettore sia già ordinato. Nel caso peggiore, basandosi sul concetto di cicli di ordinamento, riposizionerà ciclicamente tutti gli elementi effettuando il massimo numero di scritture ovvero  $N$ .

```
1 int cycle(int v[], int nmax, int spos){
2   int j, temp, change, temppos, oldtemppos;
```

```

3
4 //Inizio a ordinare la posizione
5 temp=v[ spos ];
6
7 //Valuto la posizione in cui dovrò inserire l'elemento
8 temppos = 0;
9 oldtemppos = temppos;
10 for(j=0; j<nmax; j++)
11     if(temp>v[j] && j!=spos) temppos++;
12
13 //Altrimenti inizio a riposizionare ciclicamente gli elementi
    nella loro posizione corretta
14 while(temppos != spos){
15
16     //Inserisco l'elemento nella sua posizione
17     if(temp != v[temppos]){
18         change = v[temppos];
19         v[temppos]=temp;
20         temp=change;
21     }
22
23     //Trovo la mia prossima posizione
24     oldtemppos = temppos;
25     temppos = 0;
26     for(j=0; j<nmax; j++)
27         if(temp>v[j] && j!=spos) temppos++;
28
29     //Se ho più elementi con lo stesso valore rimappo la
        posizione
30     while(temp == v[temppos])
31         if(++temppos == oldtemppos) return;
32 }
33
34
35 //Scrivo nella posizione di arrivo l'elemento salvato in temp
36 if(temp != v[temppos])
37     v[temppos]=temp;
38 }
39
40 //Ciclo principale dell'OlivatoCycleSort
41 int ocsort(int v[], int nmax){
42     int i=0;
43
44     //Scorro gli elementi nel vettore
45     while(i<nmax){
46
47         //Scorro il vettore finché c'è un ordinamento
48         while( i < nmax-1 && v[i] <= v[i+1] ) i++;
49

```

```

50     //Se ho raggiunto la fine del vettore termino
51     if( i == nmax-1) return;
52
53     cycle(v, nmax, i+1);
54     i++;
55 }
56 }
57
58 //Creo un vettore riempito di numeri casuali
59 void generate_vector(int v[], int nmax, int intervallo){
60     int i;
61
62     //Inizializzo il randomizzatore
63     srand(time(NULL));
64
65     //Inizializzo il vettore con numeri casuali
66     for(i=0;i<nmax;i++)
67         v[i]=rand() % intervallo;
68
69 }
70
71 int main(int argc, char *argv[]) {
72
73     //Recupero il numero di elementi da generare
74     int nmax;
75     nmax = atoi(argv[1]);
76     int intervallo = nmax;
77     int v[nmax];
78
79     //Genero un vettore casuale di lunghezza nmax
80     generate_vector(v, nmax, intervallo);
81
82     //Eseguo l'algoritmo
83     ocsort(v,nmax);
84 }

```

Listing 4.2: Codice C dell'implementazione dell'OlivatoCycleSort

## 4.4 Test energetici e risultati

Dai test energetici notiamo un evidente preponderanza del numero esponenziale di confronti che va ad appesantire notevolmente il carico energetico e computazionale. Nonostante questo si possono notare delle differenze abbastanza significative tra i due algoritmi, che vedono l'OlivatoCycleSort essere migliore sia dal punto di vista dell'efficienza energetica, sia dal punto di vista della durata di vita della nostra batteria di riferimento.



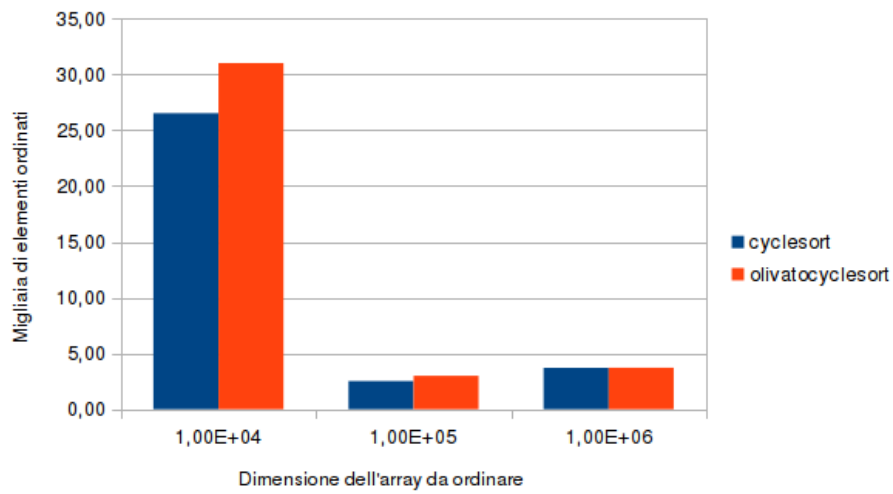


Figura 4.1: Numero totale di elementi ordinati prima del completo scaricamento della batteria di riferimento utilizzando tre differenti lunghezze di vettore.

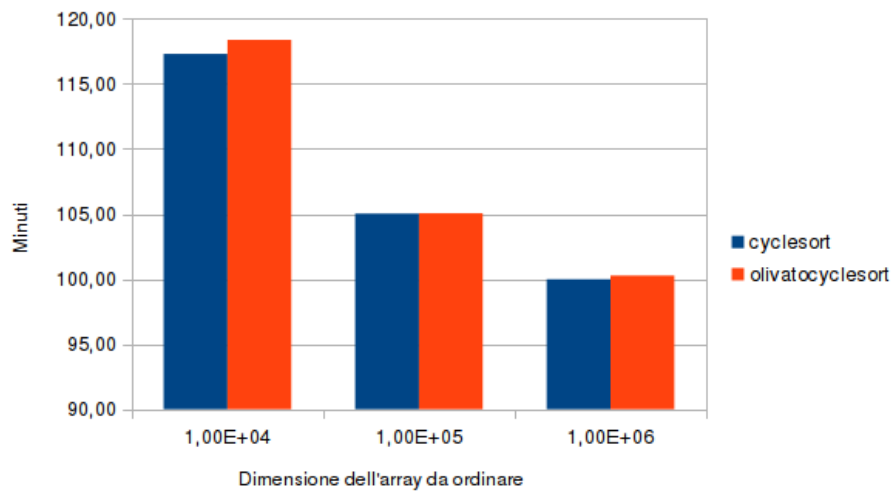


Figura 4.2: Durata della batteria di riferimento rispetto alle tre lunghezze di vettore.

## 4.5 Interesse del caso di studio

Questi due algoritmi, sebbene confrontati con molti altri algoritmi classici risultino allo stato attuale completamente inefficienti temporalmente ed energeticamente, potrebbero fare la differenza in altri contesti.

Nel caso ipotetico in cui si possano realizzare architetture con un costo di lettura e confronto pari a zero energeticamente e temporalmente[8], il costo energetico totale risulterebbe solamente legato al numero di scritture. Si possono vedere già i primi esempi di architetture orientate a questa particolare implementazione all'interno dei progetti di calcolatori quantistici[2] che userebbero il fenomeno dell'entanglement quantistico. Questa proprietà quantistica permette di leggere lo stato di una particella istantaneamente guardando la sua particella gemella anche se le due particelle sono molto distanti tra loro, in quanto ad ogni modifica della prima ne risulta una istantanea e uguale modifica della seconda.

Esiste un'altra condizione, ovvero quando il costo energetico di una scrittura è estremamente alto, nel qual caso si intuisce che il minimizzare il numero di scritture sia l'unica strategia vincente.

Un'ultima condizione particolarmente vantaggiosa potrebbe essere quella nella quale il supporto su cui si vogliano scrivere gli elementi ordinati, dopo un certo numero di scritture non possa più essere riscritto. E' chiaro che in questo caso la vita del supporto si allungerebbe rispetto ad altri algoritmi non ottimi in questo senso. Quest'ultima condizione potrebbe essere particolarmente importante per quanto riguarda i robot spaziali dotati di Memoria a Stato Solido(SSD)[19]. La condizione nello spazio dei rover consiste nell'avere con se una quantità di energia sufficiente, grazie ai pannelli solari e alle batterie al Plutonio[14], una quantità di tempo da dedicare alle operazioni effettuabili sui dati anche abbastanza lungo, visto i lunghi tempi morti, ma una quantità scarsa di risorse.

Un SSD che ha terminato il suo numero di scritture risulta infatti non più utilizzabile se non per leggere i dati rimasti in esso. In quelle condizioni estreme il Rover risulterebbe incapace di effettuare nuove misurazioni in quanto non in grado di scriverle sulla memoria e aspettare il tempo giusto per inviarle. La vita utile di un SSD potrebbe essere quindi ottimizzata proprio in questo caso, se l'ordinamento sui dati prelevati deve essere effettuato sulla memoria a stato solido.

# Capitolo 5

## Algoritmi non uniformi

### 5.1 Panoramica generale

Un'idea differente ma molto interessante risulta essere l'approccio non uniforme. Ovvero ordinare gli elementi seguendo un albero di decisione che tiene traccia della permutazione in ogni suo nodo e dove la foglia contiene l'esatto numero minimo di scambi necessario ad ordinare quella permutazione.

Questo tipo di algoritmi si chiamano non uniformi perché non è possibile determinare a priori, guardando il codice dell'algoritmo, quali saranno i passi sequenziali che l'algoritmo svolgerà per ordinare. L'algoritmo infatti procederà a salti tra le righe di codice, avendo come nodo di diramazione un confronto. L'emblema di questo tipo di algoritmo di ordinamento per il problema dell'ordinamento per confronti è il Minimum Comparison Sort. Come si evince dal nome si sta cercando quell'algoritmo che rende minimo il numero di confronti da effettuare per identificare la natura della permutazione e ordinarla di conseguenza. Per contro presentiamo ora un algoritmo deterministico che tenta[10] di raggiungere l'ottimalità nei confronti: l'Algoritmo di Ford-Johnson[9].

### 5.2 Algoritmo di Ford-Johnson (Merge-Insert)

Questo algoritmo di prefigge l'obiettivo di ordinare un gruppo di elementi effettuando il minor numero di confronti. E' composto in tre passi principali:

1. Confronto tutti gli elementi a coppie ordinandoli secondo l'ordinamento desiderato
2. Ordina poi tutti gli elementi massimi tra loro

3. Si scoprono così due elementi già ordinati, il minimo che deve essere inserito all'inizio e il massimo globale che non dovrà più essere confrontato. Successivamente si reitera sulla parte centrale della struttura dati avente gli elementi minori non confrontati tra loro che dovranno essere inseriti tra quelli massimi precedentemente già ordinati tra loro.

Questo algoritmo è stato provato in molti casi essere ottimale per il numero di confronti fino a certe lunghezze di vettore. Il caso maggiore è stato dimostrato per 47 elementi[12].

### 5.2.1 Considerazioni generali

Sebbene questo algoritmo sia stato dimostrato ottimo dal punto di vista dei confronti per alcune lunghezze di vettore ha delle caratteristiche non ottime, ovvero:

- Non effettua il minor numero di scritture, in quanto utilizza scambi e non cicli di ordinamento come visto nel capitolo precedente.
- Il caso migliore dal punto di vista dei confronti non è ottimo. E' banale notare infatti non è in grado di riconoscere proprietà transitive intrinseche negli elementi di un array e che quindi banalmente l'array già ordinato comporterebbe più di N confronti.
- Il caso pessimo non è ottimo con tutte le possibili lunghezze di vettore ma approssimativamente ottimo.
- Potrebbe comportare l'utilizzo di una struttura dati esterna complessa per effettuare efficacemente l'ordinamento

## 5.3 Algoritmo non uniforme ideale

Il vantaggi di usare un algoritmo non uniforme per affrontare il problema dell'ordinamento sono i seguenti:

1. Otteniamo un albero di confronti, ovvero un albero binario, che può essere portato ad essere ottimo per qualsiasi combinazione. L'idea è rendere minimo il percorso tra la radice, ovvero il primo confronto, e la foglia che è l'identificativo di come ordinare quella permutazione.
2. Ogni combinazione sarà una foglia dell'albero e la complessità per raggiungerla sarà la lunghezza del percorso dalla radice alla foglia.

3. Una volta identificata la singola combinazione si possono ottimizzare il numero degli scambi e quindi della scrittura per la specifica combinazione rendendo veramente minimo il numero di scritture.

Questo algoritmo permetterebbe di ordinare un gruppo di  $N$  elementi sia con il minor numero di confronti possibili sia con il minor numero di scritture, a patto di aver generato in precedenza l'albero per quella data lunghezza. Purtroppo non è teoricamente applicabile per contesti in cui si vuole ordinare un vettore di grandi dimensioni. Per capire intuitivamente questa affermazione facciamo un esempio esplicativo.

Ipotizziamo di voler avere un algoritmo non uniforme che ordina tramite albero di confronti una lunghezza pari a 15 elementi. Sicuramente dovrà avere al suo interno del codice per l'ordinamento di ogni combinazione che risiederà nelle foglie dell'albero binario. Andiamo quindi a capire quante sono le foglie: abbiamo 15 elementi, da cui le combinazioni possibili sono  $N!$ , perciò il risultato è  $15!$ . Per comprendere l'elevato numero di combinazioni e la relativa dimensione del codice che verrebbe generato, prendiamo in considerazione di spendere 1 bit per ogni combinazione e quindi solamente per identificare ogni foglia dell'albero. In questa situazione stiamo escludendo tutto il codice che permette di arrivare alle foglie dell'albero, ovvero i nodi interni e anche l'effettivo codice utile all'ordinamento ottimale per le scritture, che dovrebbe risiedere all'interno delle foglie. Il risultato calcolato verrebbe:

$$15! \text{ bit} = 1307674368000 \text{ bit} = 163459296000 \text{ byte} = 129,47 \text{ GByte}$$

Come abbiamo già detto il vero risultato risulterebbe molto maggiore di quello ottenuto. E' banale capire che dover eseguire un codice che pesi più di 130 GByte crei diversi problemi, soprattutto dal punto di vista della memoria centrale, ma anche dal punto di vista della località. L'albero per sua natura avrà dei salti tra punti di codice anche distanti tra loro, il che comporterebbe un continuo processo di swap dalla memoria centrale alla memoria secondaria di parti di codice. Aumentando il numero degli elementi anche solo di 5 unità, le cose peggiorano drasticamente arrivando ad una dimensione enorme: 240 PByte. Questo tipo di algoritmo risulta molto interessante invece nei casi in cui i vettori sono piccoli, nell'ordine dai 3 ai 6 elementi.

## 5.4 Innesto in algoritmi uniformi

Un'intuizione interessante per il nostro campo di interesse sarebbe integrare un algoritmo non uniforme di ordinamento ottimo su piccole dimensioni di array, all'interno di un algoritmo di ordinamento già noto e utilizzato anche per array di grandi dimensioni.

Gli algoritmi candidati in cui si potrebbe effettuare un innesto sono gli algoritmi basati sul concetto di divide et impera. Infatti se l'algoritmo generale ricorsivamente divide il problema dell'ordinamento in porzioni di sottoarray da ordinare possiamo, se la lunghezza del sottoarray da ordinare è abbastanza piccola, sostituirlo con l'algoritmo di ordinamento non uniforme che finisce il lavoro di ordinamento su quel piccolo sottoarray. Idealmente un algoritmo divide et impera può essere visto come un albero binario ricorsivo.

L'idea è quella di aumentare l'efficienza bloccando l'algoritmo divide et impera prima di arrivare alle foglie (che sarebbero una singola posizione nell'array), esattamente nel livello in cui dovrebbe applicarsi ricorsivamente ad una array di dimensione sufficientemente piccola per poter essere ordinato con un algoritmo non uniforme ottimo. Dovremmo ottenere un aumento di efficienza sia da un punto di vista computazionale, in quanto per un gruppo di foglie ordinate il numero di confronti effettuati è ottimo, sia dal punto di vista energetico, visto che l'algoritmo per ogni singola combinazione ha memorizzato il numero minimo di scambi per ordinare la stessa.

### 5.4.1 Algoritmo candidato all'integrazione

Ci sono due algoritmi famosi che ragionano per divide et impera:

- MergeSort
- QuickSort

Il MergeSort[11] è un algoritmo con complessità teorica ottimale  $\Theta(N \log N)$ . Ha l'enorme difetto di utilizzare molta memoria ausiliaria per poter effettuare il merge tra vettori. Questo comporta l'allocazione di memoria continua in ogni passo ricorsivo dell'algoritmo. Inoltre per sua natura dopo aver ordinato nell'array temporaneo deve ricopiare le modifiche sull'array originale, il che banalmente si intuisce non esser ottimo sulla somma totale delle scritture. Il QuickSort[13] invece seppur non essendo formalmente ottimo dal punto di vista computazionale, in quanto possiede un caso (estremamente raro) in cui la complessità è dell'ordine di  $O(N^2)$ , il caso medio è  $O(N \log_2 N)$ . Il vantaggi principali sono che:

1. E' un algoritmo In-place che quindi non alloca memoria secondari. Nella sua versione ricorsiva, deve tener traccia delle chiamate ricorsive spendendo  $\log_2 N$  memoria. Esiste comunque la versione iterativa che elimina anche questo piccolo uso della memoria, che diventa quindi costante.

2. Possiede una elevata località del codice, in quanto le chiamate alla funzione, che siano ricorsive o meno, possono essere scritte abbastanza vicine tra loro. La totalità della funzione compilata ha dimensioni veramente ridotte.
3. Possiede una buona località sulla struttura dati utilizzata. Infatti utilizzando la strategia divide et impera suddivide i blocchi da ordinare in porzioni più piccole che banalmente per essere ordinate potranno risiedere tutte in memoria completamente senza dover effettuare ulteriori swap.
4. Grazie alle sue prestazioni già collaudate nel tempo è l'algoritmo base utilizzato dall'Introsort. L'Introsort[6] è una variante che combina insieme tre algoritmi: Quicksort - HeapSort - InsertionSort. Questo algoritmo è quello implementato nella libreria standard dell' ANSI C richiamabile tramite la funzione qsort[4].

In definitiva l'algoritmo scelto come base per l'integrazione con gli algoritmi non uniformi sarà il QuickSort. L'interesse maggiore risiederà nei confronti tra le varie implementazioni modificate e testate e il qsort.

## 5.5 Implementazioni integrate

Per poter integrare all'intero del Quicksort delle funzioni non uniformi di ordinamento abbiamo prima prodotto le funzioni e poi abbiamo modificato il Quicksort affinché le potesse utilizzare. Come abbiamo già visto la lunghezza dell'array non potrà essere troppo elevata, perciò è stato deciso di produrre tre versioni:

1. **Threesort:**  
Una funzione di ordinamento non uniforme che ordina tre elementi e effettua un numero ottimo di scambi tra gli elementi per ordinare la permutazione.
2. **Foursort:**  
Una funzione identica alla precedente solamente creata per una array di quattro elementi.
3. **Sixsort:**  
Una funzione di ordinamento non uniforme che si basa sul Threesort e sul Foursort. Questa funzione non è ottima da un punto di vista dei confronti e degli scambi ma è sicuramente molto vicina all'ottimo. E' comunque

stata presa in considerazione perché è dotata di un codice abbastanza contenuto rispetto, alla stessa implementata ad albero analoga alle due precedenti.

Facciamo notare che tutte e tre sfruttano gli scambi come metodo di ordinamento reale e che quindi non potranno essere ottime dal punto di vista delle scritture. Di seguito i codici principali delle funzioni implementate.

```

1 //Funzione che scambia due elementi di un vettore
2 void swap(int v[], int a, int b){
3     int c = v[a];
4     v[a] = v[b];
5     v[b] = c;
6 }

```

Listing 5.1: Codice C della funzione di swap

```

1 //Funzione che ordina con il minor numero di confronti un
  vettore di lunghezza 3
2 void threesort(int v[], int start){
3     int a = start;
4     int b = start+1;
5     int c = start+2;
6
7     if(v[a]<v[b]){
8         if(v[b]>v[c]){
9             if(v[a]<v[c]){
10                swap(v,b,c);
11            }else{
12                swap(v,b,c);
13                swap(v,a,b);
14            }
15        }
16    }else{
17        if(v[c]<v[b]){
18            swap(v,a,c);
19        }else{
20            if(v[a]<v[c]){
21                swap(v,a,b);
22            }else{
23                swap(v,a,b);
24                swap(v,b,c);
25            }
26        }
27    }
28 }

```

Listing 5.2: Codice C della funzione di ordinamento Threesort



```
1 //Funzione che ordina con il minor numero di confronti un
   vettore di lunghezza 4
2 void foursort(int v[], int start){
3
4     int a=start+0;
5     int b=start+1;
6     int c=start+2;
7     int d=start+3;
8
9     if(v[a]<v[b]){
10         if(v[b]<v[c]){
11             if(v[c]>v[d]){
12                 if(v[b]<v[d]){
13                     swap(v,c,d);
14                 }else{
15                     if(v[a]<v[d]){
16                         swap(v,c,d);
17                         swap(v,b,c);
18                     }else{
19                         swap(v,c,d);
20                         swap(v,b,c);
21                         swap(v,a,b);
22                     }
23                 }
24             }else{
25                 return;
26             }
27         }else{
28             if(v[c]>v[d]){
29                 if(v[d]<v[a]){
30                     if(v[a]<v[c]){
31                         swap(v,b,d);
32                         swap(v,a,b);
33                     }else{
34                         swap(v,b,c);
35                         swap(v,c,d);
36                         swap(v,a,c);
37                     }
38                 }else{
39                     swap(v,b,d);
40                 }
41             }else{
42                 if(v[b]<v[d]){
43                     if(v[a]<v[c]){
44                         swap(v,b,c);
45                     }else{
46                         swap(v,a,c);
47                         swap(v,b,c);
48                     }
49                 }
50             }
51         }
52     }
53 }
```

```

49         }else{
50             if(v[a]<v[c]){
51                 swap(v,b,c);
52                 swap(v,c,d);
53             }else{
54                 if(v[a]<v[d]){
55                     swap(v,b,c);
56                     swap(v,a,b);
57                     swap(v,c,d);
58                 }else{
59                     swap(v,a,c);
60                     swap(v,b,d);
61                 }
62             }
63         }
64     }
65 }
66
67 }
68
69 }else{
70     if(v[a]<v[c]){
71         if(v[c]<v[d]){
72             swap(v, a, b);
73         }else{
74             if(v[a]<v[d]){
75                 swap(v, a, b);
76                 swap(v, c, d);
77             }else{
78                 if(v[b]<v[d]){
79                     swap(v,a,b);
80                     swap(v,b,d);
81                     swap(v,c,d);
82                 }
83             }else{
84                 swap(v,a,c);
85                 swap(v,a,d);
86             }
87         }
88     }
89 }else{
90     if(v[a]<v[d]){
91         if(v[b]<v[c]){
92             swap(v,a,b);
93             swap(v,b,c);
94         }else{
95             swap(v,a,c);
96         }
97     }

```

```

98     }else{
99         if(v[b]<v[c]){
100             if(v[c]<v[d]){
101                 swap(v,a,b);
102                 swap(v,b,c);
103                 swap(v,c,d);
104             }else{
105                 if(v[b]<v[d]){
106                     swap(v,a,b);
107                     swap(v,b,d);
108                 }else{
109                     swap(v,a,d);
110                 }
111             }
112         }else{
113             if(v[d]<v[c]){
114                 swap(v,a,d);
115                 swap(v,b,c);
116             }else{
117                 if(v[b]<v[d]){
118                     swap(v,a,c);
119                     swap(v,c,d);
120                 }else{
121                     swap(v,b,c);
122                     swap(v,a,b);
123                     swap(v,b,d);
124                 }
125             }
126         }
127     }
128 }
129 }
130 }

```

Listing 5.3: Codice C della funzione di ordinamento Foursort

```

1 //Funzione che ordina sei elementi usando il threesort e il
  foursort
2 void sixsort(int v[], int start){
3
4     //Primo ordino i sei elementi per gruppi da tre
5     threesort(v,start);
6     threesort(v,start+3);
7
8     if(v[start+2]<v[start+3]) return;
9
10    //Poi trovo il massimo e il minimo del vettore
11    if(v[start]>v[start+3])
12        swap(v,start,start+3);
13

```

```

14  if(v[start+2]>v[start+5])
15      swap(v, start+2, start+5);
16
17  //Poi ordino gli elementi centrali
18  foursort(v, start+1);
19 }

```

Listing 5.4: Codice C della funzione di ordinamento Sixsort

La funzione Quicksort sarà opportunamente modificata per ogni tipo di funzione che si voglia utilizzare. Di seguito daremo come esempio una delle tre varianti di utilizzo:

```

1  //Funzione di ordinamento che usa il quick sort modificato alla
   base
2  void nqs_six( int a[] , int l, int r){
3      int j;
4
5      if( l < r ){
6
7          j = partition( a, l, r);
8
9          if(j-l == 6)
10             sixsort(a, l);
11         else
12             nqs_six( a, l, j-1);
13
14         if(r-j+1 == 6)
15             sixsort(a, j);
16         else
17             nqs_six( a, j+1, r);
18     }
19 }
20 }
21
22 //Funzione che esegue la partizione dell'array
23 int partition( int data[] , int first , int last) {
24     int pivot = data[first];
25     int left = first;
26     int right = last;
27     int t;
28
29     while ( left < right ){
30         while (data[left] <= pivot && left < right) left++;
31         while (data[right] > pivot) right--;
32         if(left < right){
33             t = data[left];
34             data[left] = data[right];
35             data[right] = t;
36         }

```

```
37  }  
38  
39  t = data[ first ];  
40  data[ first ] = data[ right ];  
41  data[ right ] = t;  
42  
43  return right;  
44 }
```

Listing 5.5: Codice C del Quicksort modificato per la Sixsort

### 5.5.1 Funzione non uniforme avanzata

Questa idea avanzata riguardo la creazione di un nuovo tipo di funzione non uniforme si è raggiunta volendo ottenere da questa alcuni risultati significativi:

1. Autogenerazione del codice della funzione o di parti di esso per valori di N arbitrari.
2. Numero di scritture durante l'operazione di ordinamento finale minimo
3. Dimensione relativamente più compatta del codice della funzione anche per valori di N non piccoli.

Il tipo di funzione risultate è stato chiamato ***KeyHardcodedSort***.

#### 5.5.1.1 Idea dell'algoritmo avanzato

L'algoritmo si basa sull'idea di dare un codice binario ad una permutazione tramite una serie di confronti. Una volta che è stato ottenuto il codice univoco di quella permutazione è possibile ordinarla eseguendo il minor numero di scritture, utilizzando l'idea dei cicli di ordinamento. In pratica per tutte le possibili permutazioni di quella lunghezza si sono registrati in un vettore gli esatti cicli di ordinamento che eseguirebbe un CycleSort o un OlivatoCycleSort se dovesse ordinare quella permutazione. Essendo stati memorizzati già i cicli, per eseguirli basta seguire il ciclo di sostituzioni identificato dal susseguirsi di posizioni e si ottiene così il vettore ordinato con il minor numero di scritture. Più chiaramente i concetti sono:

1. Identificare la singola permutazione con un codice binario univoco dove il generatore sarà una algoritmo per confronti
2. Una volta che la singola permutazione è stata identificata la andremo ad ordinare con dei cicli di ordinamento precedentemente precalcolati da un OlivatoCycleSort.

E' evidente come questo algoritmo effettui il minor numero di scritture possibili, anche se non si può identificare facilmente la sua complessità. Per la natura modulare dell'algoritmo la complessità risiede in due operazioni:

1. La parte di generazione del codice binario
2. La parte di ricerca dei cicli di ordinamento precalcolati nel vettore associati a quel codice binario.

Vedremo che tale complessità per numeri inferiori di 9 rimarrà compresa tra  $O(N + \log_2 N)$  e  $O(2N \log_2 N)$ .

### 5.5.1.2 Descrizione dettagliata

Come abbiamo visto l'algoritmo si racchiude in due parti. Ci sono però delle operazioni intermedie che permettono di gestire i dati in modo tale da renderli più facilmente accessibili e manipolabili. Di seguito si spiega in dettaglio:

1. Data una permutazione di  $N$  oggetti in ingresso si cerca di determinare il codice binario identificativo univoco della stessa. Per fare ciò si utilizza un algoritmo che effettua esattamente  $N * (N - 1)/2$  confronti. Questo algoritmo è stato ideato per controllare prima le proprietà di transitività interne all'array. Mentre si sta generando il codice binario l'algoritmo passa i bit ottenuti ad una funzione atta alla ricerca della chiave della permutazione.
2. Una volta ottenuta la chiave della permutazione, ovvero l'indice nell'array dei cicli di ordinamento da dover effettuare, si applicano le sostituzioni cicliche ordinando l'array.

Per ridurre le complessità e compattare il codice la funzione che deve cercare la chiave a partire dai bit del codice generati a run-time, contiene al suo interno un albero binario generato come segue:

1. Trovo tutte i i possibili codici binari associati a tutte le possibili permutazioni
2. Applico un'euristica sui codici binari identificando la lunghezza minima per cui ogni particolare codice risulti univoco rispetto agli altri.
3. Creo un albero binario a partire dai codici binari, avente quindi tutti i possibili percorsi che rappresentano la totalità dei codici binari ottimizzati. Nelle foglie di ogni percorso, e quindi del singolo codice binario, associo la chiave corretta per quella permutazione.

4. L'albero binario generato al punto precedente è linearizzato e schiacciato all'interno di un vettore di interi.

Questa funzione ricercherà su un albero binario linearizzato di profondità massima pari alla lunghezza massima del codice binario in ingresso, ovvero  $N(N-1)/2$ . Dunque la ricerca richiederà al massimo  $\log_2(N(N-1)/2)$  confronti.

### 5.5.1.3 Implementazione e integrazione dell'algoritmo

Seppur questo algoritmo risulti molto più compatto dei precedenti sono state generate implementazioni accettabili dimensionalmente solo fino ad array di lunghezza 7. Anch'esso è stato inserito come base del Quicksort con lo scopo di aumentarne l'efficienza energetica. Di seguito è rappresentato l'esempio di implementazione con un sottoarray di lunghezza 3.

```

1 int chiavi[11][2] =
    {{1,6},{2,3},{-1,-1},{4,5},{-2,-2},{-3,-3},{7,10},
2     {8,9},{-4,-4},{-5,-5},{-6,-6}};
3
4 int cicli[6][5] = {{0,2,0,-1}, {0,2,1,0,-1}, {0,1,0,-1},
    {0,1,2,0,-1},
5     {1,2,1,-1}, {-1} };
6
7 int NMAXK = 3;
8
9 //Funzione che recupera la chiave della combinazione e il ciclo
10 // da effettuare per ordinarla
11 int getkey(int v[], int start){
12     int lvett=NMAXK;
13     int btf;
14     int i,j;
15     int chiave=0;
16
17     for(i=1;i<lvett;i++){
18         for(j=0;j<lvett-i;j++){
19
20             //Trovo il bit della chiave da cercare
21             btf = v[start+j]<=v[start+j+i]?1:0;
22             //printf("btf: %i\n",btf);
23
24             //Recupero il puntatore interno al vettore al prossimo bit
25             chiave = chiavi[chiave][btf];
26             //printf("chiave: %i\n",chiave);
27
28             //Se il puntatore punta ad un valore negativo allora lo
                ritorno

```

```

29         if(chiavi[chiave][0] < 0) return (chiavi[chiave][0] *
30             -1)-1;
31     }
32 }
33 return -1;
34 }
35
36 //Ordino la combinazione usando i cicli
37 void sortkc(int v[], int start, int chiave){
38     int i, startpos, temp, val;
39     int lvett = NMAXK;
40
41     if(cicli[chiave][0]==-1) return;
42
43     startpos = cicli[chiave][0];
44     val = v[start + startpos];
45
46     for(i=1; cicli[chiave][i]!=-1; i++){
47
48         //Se sto ricominciando un ciclo
49         if(startpos == -1){
50             startpos=cicli[chiave][i];
51             val = v[start + startpos];
52         }
53
54         //Se sono dentro un ciclo normale
55         if(startpos != cicli[chiave][i]){
56             temp=v[start + cicli[chiave][i]];
57             v[start + cicli[chiave][i]]=val;
58             val=temp;
59             startpos=cicli[chiave][i];
60         }else{
61             //Ho finito il ciclo
62             v[start + cicli[chiave][i]]=val;
63             startpos=-1;
64         }
65     }
66 }
67 }
68
69 //Funzione di ordinamento che usa il quick sort modificato alla
    base
70 void quicksort_keyhardcoded( int a[], int l, int r){
71     int j;
72
73     if( l < r ){
74
75         // divide and conquer

```



```
76     j = partition( a, l, r);
77
78     if(j-1 == NMAXK)
79         sortkc(a,l, getkey(a,l));
80     else
81         quicksort_keyhardcoded( a, l, j-1);
82
83     if(r-j+1 == NMAXK)
84         sortkc(a,j, getkey(a,j));
85     else
86         quicksort_keyhardcoded( a, j+1, r);
87
88 }
89 }
90
91 int partition( int data[], int first, int last) {
92     int pivot = data[first];
93     int left = first;
94     int right = last;
95     int t;
96
97     while (left < right){
98         while (data[left] <= pivot && left < right) left++;
99         while (data[right] > pivot) right--;
100         if(left < right){
101             t = data[left];
102             data[left] = data[right];
103             data[right] = t;
104         }
105     }
106
107     t = data[first];
108     data[first] = data[right];
109     data[right] = t;
110
111     return right;
112 }
113
114 //Creo un vettore riempito di numeri casuali
115 void generate_vector(int v[], int nmax, int intervallo, int
    reset){
116     int i;
117
118     //Inizializzo il randomizzatore
119     if(reset)
120         srand(time(NULL));
121
122     //Inizializzo il vettore con numeri casuali
123     for(i=0;i<nmax;i++)
```

```
124     v[i]=rand() % intervallo;
125
126 }
127
128 int main(int argc, char *argv[]) {
129
130     //Recupero il numero di elementi da generare
131     int nmax;
132     nmax = atoi(argv[1]);
133     int intervallo = nmax;
134     int v[nmax];
135
136     //Genero il vettore disordinato
137     generate_vector(v, nmax, intervallo, 1);
138
139     //Ordino il vettore con l'algoritmo da testare
140     quicksort_keyhardcoded(v, 0, nmax-1);
141
142     return 0;
143 }
```

Listing 5.6: Codice C del KeyHardCodedSort per 3 elementi integrato nel Quicksort

## 5.6 Test e Grafici

Di seguito sono riportati i test effettuati sulle varie implementazioni degli algoritmi. L'aspetto più interessante riguarda le problematiche di confronto dell'efficienza e della durata di vita della batteria di riferimento da 5V e 1000mAh. Sono stati creati dei grafici il più chiari possibili, dove si evidenziano sia i miglioramenti rispetto ad una implementazione di riferimento sia i peggioramenti rispetto alla stessa.

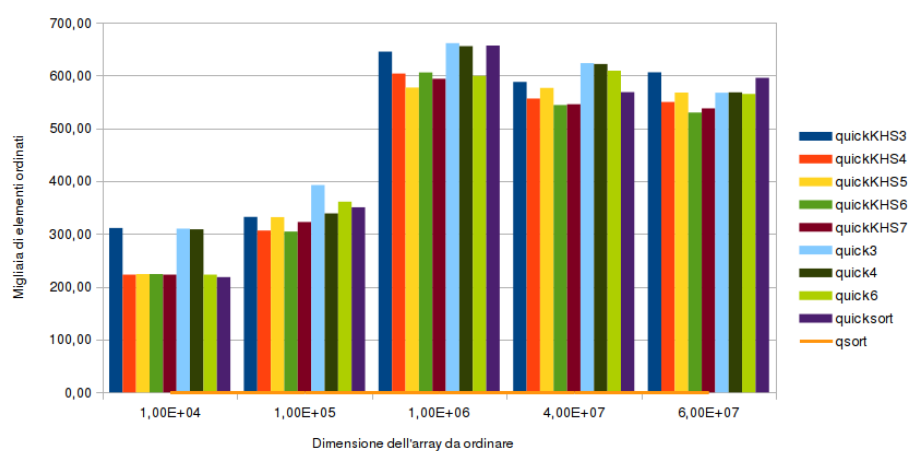


Figura 5.1: Numero totale di elementi ordinati prima del completo scaricamento della batteria di riferimento rispetto alla funzione qsort dell' ANSI C.

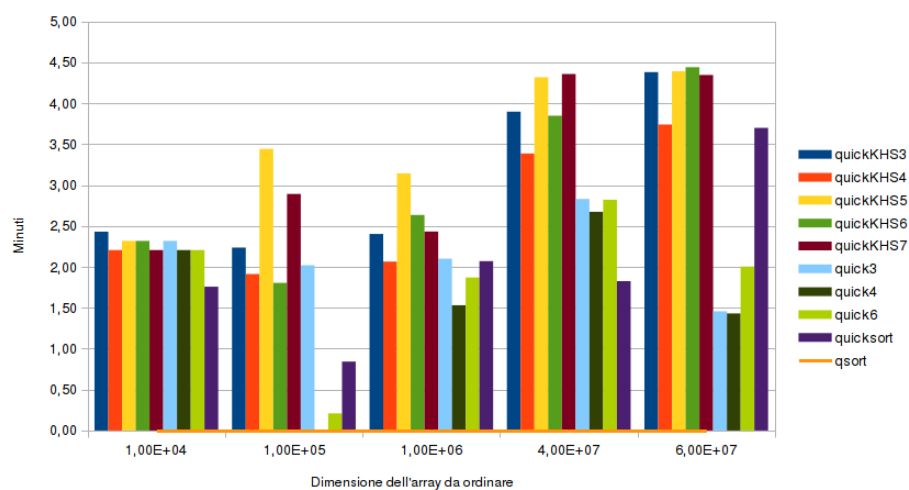


Figura 5.2: Durata della batteria di riferimento rispetto alla funzione qsort dell' ANSI C.

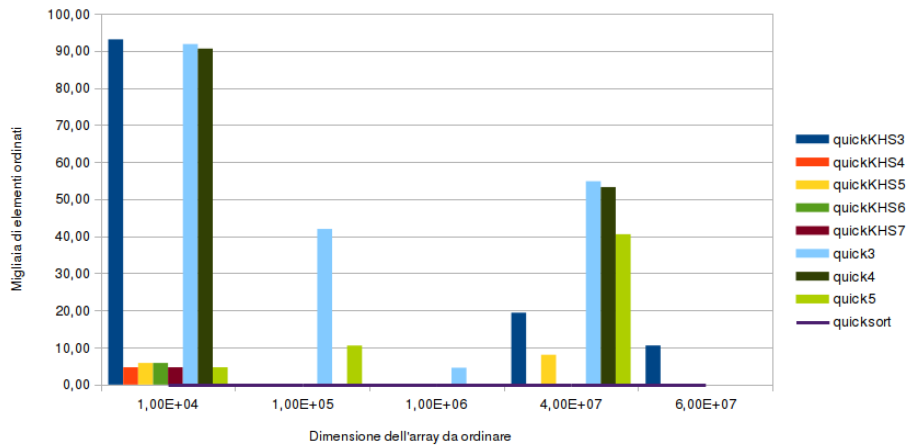


Figura 5.3: Numero totale di elementi ordinati prima del completo scaricamento della batteria di riferimento rispetto al quicksort.

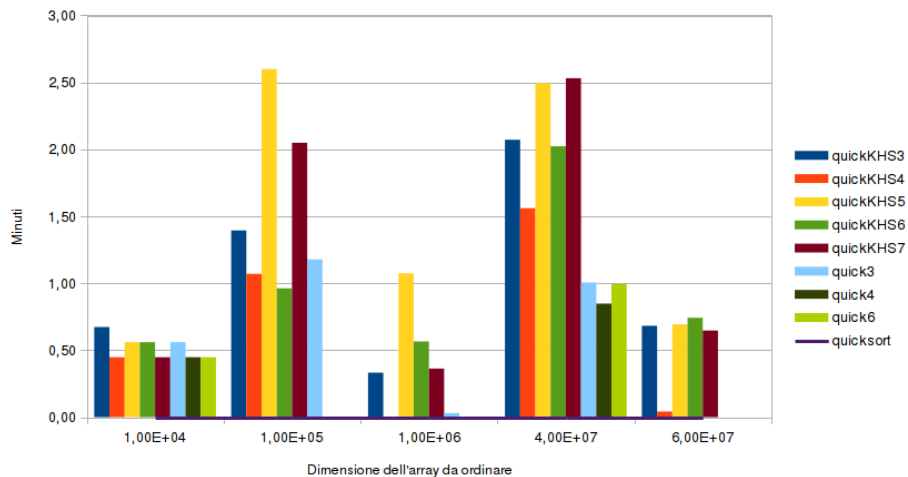


Figura 5.4: Differenze nella durata della batteria di riferimento rispetto al quicksort.

Si nota chiaramente che per quanto riguarda la vita della batteria, l'implementazione del Quicksort con KeyHardcodedSort a 5 elementi (quickKHS5) risulta essere la migliore in assoluto, sia nei confronti dell'implementazione qsort, sia del Quicksort normale. Dal punto di vista dell'efficienza computazionale invece notiamo che gli algoritmi proposti anche se molti migliorano la durata di vita della batteria, pochi riescono ad essere anche più efficienti, ovvero riescono ad ordinare più elementi degli algoritmi standard e contempo-

raneamente consumare anche meno energia. L'unico ad riuscire nell'intento e degno di nota è l'implementazione Quicksort + Threesort (quick3) che è in grado di ordinare più elementi degli algoritmi standard per quasi tutte le differenti lunghezze ma è anche in grado di consumare meno energia allungando la durata di vita della batteria.



# Capitolo 6

## Conclusioni

### 6.1 Conclusione e prospettive future

Abbiamo visto nei capitoli precedenti come lo studio del consumo energetico in ambito software dia dei risultati evidenti e ha permetta miglioramenti algoritmici altrimenti non considerati.

Il primo risultato è stata la riconferma dello Standard swap come l'algoritmo di scambio più energeticamente efficiente e consigliato per quasi tutti gli utilizzi.

Il secondo risultato interessante riguarda la ricerca degli algoritmi del minor numero di scritture, con la scoperta di un algoritmo ideato in proposito, il CycleSort e un'anteprima inedita di un algoritmo che cerca di migliorarlo, l'OlivatoCycleSort. L'approfondimento su questo caso di studio ne fa capire l'importanza e gli aspetti sconosciuti che sono decisamente più proiettati in una visione futura, sia algoritmica ma soprattutto architettuale.

Il terzo e più interessante risultato presenta i miglioramenti effettuati nell'algoritmo di ordinamento di riferimento per il linguaggio ANSI C. Non solo è possibile affermare che una modifica algoritmica che tenga conto dei parametri di incidenza sul consumo è evidente e misurabile, ma che può in alcuni casi fare una differenza marcata nella durata di vita della batteria.

L'approccio iniziale si è riferito all'utilizzo di algoritmi non uniformi come fonte di miglioramento energetico e dell'efficienza, integrati nella struttura algoritmica di riferimento, il Quicksort. Successivamente, a causa della necessità e della curiosità di testare nuove forme di algoritmi uniformi, che avessero un codice di dimensione ridotta rispetto ai precedenti e che potessero essere generati in modo automatico, si sono presentate delle nuove funzioni non uniformi di ordinamento che sfruttano strutture dati particolari e i risultati precalcolati di altri algoritmi per fornire una alternativa valida a quelli classici.

I grafici sono esplicativi e chiarificano il lavoro svolto dimostrando un'evidente riconoscimento degli sforzi e della significatività dei risultati.

Le versioni testate sono state molte, sia per dare un ampio spettro del risultato che si ottiene attraverso le modifiche, sia per trovare un algoritmo modificato candidato ad essere più efficiente su tutti i fronti. Quest'ultimo potrà diventare di interesse implementativo e strutturale in quegli algoritmi che si basano sull'ordinamento, o sulla ripetizione ciclica di ordinamenti, e facendo notare subito risultati evidenti sul consumo energetico totale.

E' importante ricordare che il lavoro svolto e descritto in questo testo potrebbe far nascere molte considerazioni, sia su come si sono svolti i test, sia sul dispositivo utilizzato, sia su ulteriori idee di modifica e miglioramento degli algoritmi. Si è consapevolmente limitato il contenuto alle informazioni essenziali e le molte considerazioni potrebbero essere riprese più avanti, perché l'argomento se trattato con la assoluta certezza avrebbe comportato una spesa di tempo e risorse incompatibili con un privato autofinanziamento. Nonostante ciò si è orgogliosi e convinti che i risultati ottenuti siano rilevanti e convincenti per un'ulteriore investimento sulla ricerca algoritmica come chiave dello sviluppo energetico futuro.



# Bibliografia

- [1] *Cache miss*. 2015. URL: [https://en.wikipedia.org/wiki/CPU\\_cache#Cache\\_miss](https://en.wikipedia.org/wiki/CPU_cache#Cache_miss).
- [2] *Calcolatore quantistico*. 2015. URL: [https://en.wikipedia.org/wiki/Quantum\\_computing](https://en.wikipedia.org/wiki/Quantum_computing).
- [3] *Cyclesort*. 2015. URL: [https://en.wikipedia.org/wiki/Cycle\\_sort](https://en.wikipedia.org/wiki/Cycle_sort).
- [4] *Funzione qsort ANSI C*. 2015. URL: <https://en.wikipedia.org/wiki/Qsort>.
- [5] B. K. Haddon. «Cycle-Sort: A Linear Sorting Method». In: *The Computer Journal* 33 (1990). URL: <http://comjnl.oxfordjournals.org/content/33/4/365.full.pdf+html>.
- [6] *Introsort*. 2015. URL: <https://en.wikipedia.org/wiki/Quicksort>.
- [7] Donald E. Knuth. *The art of computer programming volume 3 (Sorting and Searching)*. Addison Wesley Pub Co Inc, 1998.
- [8] *Landauer's principle*. 2015. URL: [https://en.wikipedia.org/wiki/Landauer%27s\\_principle](https://en.wikipedia.org/wiki/Landauer%27s_principle).
- [9] Jr. Lester R. Ford e Selmer M. Johnson. «A Tournament Problem». In: *The American Mathematical Monthly* 66 (1959). URL: <http://www.jstor.org/stable/2308750>.
- [10] Glenn K. Manacher. «The Ford-Johnson Sorting Algorithm Is Not Optimal». In: *Journal of the ACM* 26 (1979). URL: <http://dl.acm.org/citation.cfm?id=322145>.
- [11] *Mergesort*. 2015. URL: [https://en.wikipedia.org/wiki/Merge\\_sort](https://en.wikipedia.org/wiki/Merge_sort).
- [12] Marcin Peczarski. «The Ford-Johnson algorithm still unbeaten for less than 47 elements». In: *Information Processing Letters* 101 (2007). URL: <http://www.sciencedirect.com/science/article/pii/S0020019006002742>.

- [13] *Quicksort*. 2015. URL: <https://en.wikipedia.org/wiki/Introsort>.
- [14] *Radioisotope Power System*. National Aeronautics e Space Administration. 2015. URL: <http://solarsystem.nasa.gov/rps/home.cfm>.
- [15] *Raspberry Pi 2 Model B*. 2015. URL: <https://www.raspberrypi.org/products/raspberry-pi-2-model-b/>.
- [16] *Raspbian Jessie*. 2015. URL: <https://www.raspberrypi.org/blog/raspbian-jessie-is-here/>.
- [17] *Standard Swap*. 2015. URL: [https://en.wikipedia.org/wiki/Standard\\_swap](https://en.wikipedia.org/wiki/Standard_swap).
- [18] *Swap through addition and subtraction*. 2015. URL: [https://en.wikipedia.org/wiki/Swap\\_\(computer\\_science\)#Swap\\_through\\_addition\\_and\\_subtraction](https://en.wikipedia.org/wiki/Swap_(computer_science)#Swap_through_addition_and_subtraction).
- [19] Guy Webster. *Computer Swap on Curiosity Rover*. National Aeronautics e Space Administration. 2013. URL: [http://www.nasa.gov/mission\\_pages/msl/news/msl20130228.html](http://www.nasa.gov/mission_pages/msl/news/msl20130228.html).
- [20] *Xor Swap*. 2015. URL: [https://en.wikipedia.org/wiki/XOR\\_swap\\_algorithm](https://en.wikipedia.org/wiki/XOR_swap_algorithm).