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 una questione 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. In questa tesi ci siamo prefissi di esplorare se e quali risparmi energetici fosse possibile ottenere in un contesto base e predominante quale quello degli algoritmi di ordinamento. Rispetto ai precedenti e recentissimi studi in argomento, di cui diamo rassegna, combiniamo metodici studi sperimentali e la proposta e tentativi di modifiche per i più classici algoritmi sequenziali. 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. Alcune direzioni e tentativi infuttuosi sperimentali vengono comunque documentati nella tesi. Per altri accorgimenti che abbiamo introdotto, i risultati dei test evidenziano chiaramente come la loro adozione conduca a ridurre i consumi energetici, e suggerisce che una scelta oculata del software e una sua ottimizzazione porta all'allungamento del tempo di vita delle batterie. La tesi cerca inoltre di quantificare quanto anche piccoli risparmi energetici possano risultare significativi e tende quindi ad evidenziare come una revisione degli studi algoritmici di base, opportunamente coniugata in chiave sperimentale, possa trovare una sua giustificazione su un tavolo di stampo tecnologico-commerciale quale quello mobile.

# Indice

# Elenco delle figure

# Listings

# Capitolo 1

# Dispositivo utilizzato e ambiente di test

## 1.1 Introduzione

Negli ultimi anni si è 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 omogena in tutte le sue componenti. Un esempio può essere riscontrato nel ritmo di miglioramento nelle nuove versioni software e delle nuove funzionalità hardware, che non sono state accompagnate da un adeguato miglioramento in termini di efficienza energetica. Si è 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. Inizia ad essere sempre più 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, sembra ancora distante e i limiti hardware col passare degli anni si fanno sempre più stringenti e costosi.

Un'idea innovativa potrebbe essere il cambiamento del punto di vista e ambito di ricerca, esaminando l'altra faccia della medaglia: il Software. Le domande che potrebbero sorgere 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, esaminando soluzioni già esistenti e 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 essere equipaggiato con 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 andrebbero 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 [36]. 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 $\times$ 56.5 mm
- Memoria di massa: microSD
- Consumo massimo stimato: 800 mA (4 Watt)

Si nota che 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 [37] 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.

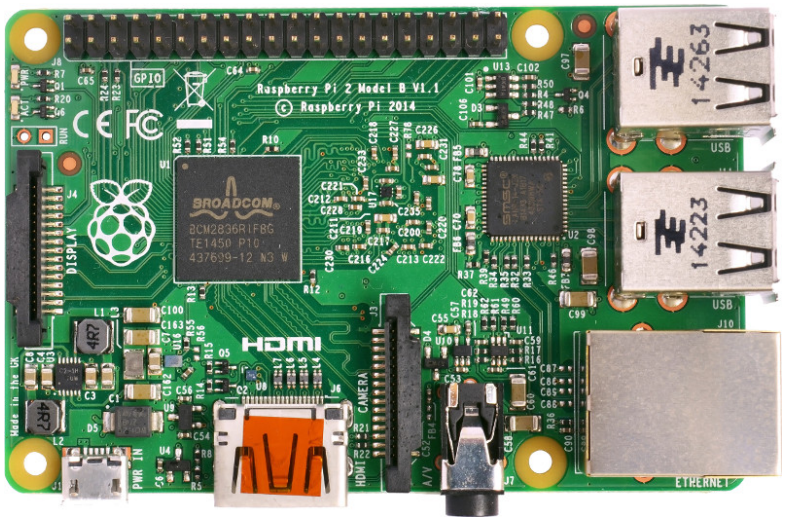


Figura 1.1: Raspberry Pi 2 Model B

## 1.3 Algoritmi scelti

Per rendere i test significativi anche per gli aspetti algoritmici si è ritenuto più opportuno andare ad indagare il consumo e un suo possibile miglioramento nell'ambito degli Algoritmi di Ordinamento. I motivi sono:

- Facilità di implementazione e modifica di tali algoritmi.
- L'ambiente di test e di debug di una modifica risulta estremamente affidabile.
- Il codice sorgente risulta generalmente contenuto, quindi facilmente caricabile nella sua interezza nella memoria centrale.

- Questi algoritmi 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 è ritenuto opportuno analizzare solo quelli che avessero avuto un'importanza e un'efficacia in una loro possibile modifica utile e significativa, tralasciando 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 [26] e Quicksort [33]**
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 [15, 24] e OlivatoCycleSort**
Questi algoritmi hanno come idea di base la minimizzazione del numero di scritture durante il processo di ordinamento. Seppur avendo una complessità quadratica e generalmente non risultino di interesse confrontati con altri algoritmi standard, potrebbero esistere dei contesti in cui un loro utilizzo possa risultare particolarmente vantaggioso.

## 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 5.2:***
   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. La versione del compilatore è la 5.2 [21], la più recente versione attualmente disponibile. I risultati sperimentali sono stati valutati utilizzando questa specifica versione, l'utilizzo di altre versioni del compilatore (esempio la 4.8 [20]) potrebbero portare a risultati differenti che in questo testo non saranno analizzati.

Ambiente Hardware:

1. ***Raspberry Pi 2 model B:***
   Il dispositivo descritto in dettaglio nella sezione 1.2.

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.

### 1.4.1 I Testbed utilizzati

Il Testbed, ovvero l'ambiente di lavoro utilizzato per i test, in qualsiasi ricerca che debba analizzare anche aspetti empirici è di fondamentale importanza. Nel nostro caso si possono identificare due Testbed:

- Il primo per i test di natura temporali riguardanti il tempo di esecuzione di un particolare algoritmo.

- Il secondo per i test di natura energetica durante l'esecuzione dell'algoritmo stesso.

#### 1.4.1.1 Testbed temporale

L'ambiente di test utilizzato per l'analisi del tempo di esecuzione di un particolare algoritmo è composto da alcuni comandi offerti dal sistema operativo Linux di Raspberry pi 2, in particolare Raspbian una versione di Debian. Questo sistema operativo offre due comandi fondamentali:

1. Il comando **time** [42] che restituisce il tempo di esecuzione di un processo calcolato in millisecondi.

2. Il comando **ulimit** [17] con l'opzione **-s** che permette di impostare la quantità massima in Kbyte di stack utilizzabile da un processo eseguito in quella shell.

Grazie al primo comando potremmo ottenere un'analisi del tempo di esecuzione accurata affidandoci a primitive Linux ampiamente ottimizzate e affidabili. Il secondo comando risulta utile nella misura in cui si voglia estendere la capacità degli algoritmi di ordinamento di manipolare array di interi molto grandi. Un errore immediato riscontrabile senza l'utilizzo preventivo di tale comando è quello relativo alla fuoriuscita della dimensione dello stack disponibile.

Volendo trovare quali siano gli effetti temporali ed energetici delle modifiche apportate agli algoritmi, l'aumentare notevolmente la dimensione dell'array da ordinare rende queste differenze molto più marcate, riducendo gli errori di misura che possono essere introdotti nei test.

#### 1.4.1.2 Testbed energetico

L'ambiente che ha permesso di effettuare le misurazioni energetiche è più complesso e articolato. Di seguito si descrivono i vari aspetti:

1. Gli algoritmi sono eseguiti sulla Raspberry Pi 2 attraverso una connessione remota via WLAN utilizzando un computer portatile. La comunicazione avviene tramite il protocollo SSH, privo di interfaccia grafica.

2. Raspberry Pi 2 deve essere dotato di antenna USB Wifi(Dongle Wifi), per permettere il collegamento alla WLAN(Wireless LAN), nel nostro caso una antenna D-Link.

3. La Wireless LAN è generata da un Router Wifi. Nella stessa sono collegati due dispositivi: la Raspberry Pi 2 e il computer portatile che impartisce i comandi.

4. La Raspberry Pi 2 è alimentata tramite un cavo microUSB, collegato ad un alimentatore USB da 5 Volt e 1 Ampere.

5. L'alimentatore USB è connesso ad una prolunga a doppia presa. La prolunga a doppia presa è una prolunga modificata in modo da ottenere una nuova presa a metà tra la spina e la presa originale a cui è collegato l'alimentatore USB.

6. Un amperometro di precisione in corrente alternata e collegato alla nuova presa che risulta quindi collegato in serie all'alimentatore USB della Raspberry Pi 2. Il fondoscala è settato sulla misurazione dei milliAmpere.

7. La prolunga a doppia presa è collegata direttamente ad una presa a muro che fornisce 220 Volt alternati.

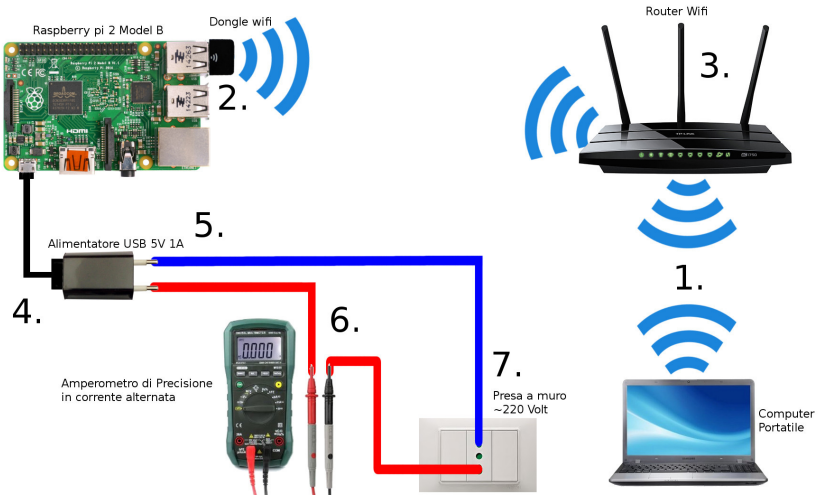


Figura 1.2: Testbed utilizzato per le misurazioni energetiche

### 1.4.2 Procedura di Test di un algoritmo

Una volta scelti, progettati e realizzati i Testbed si passa alle misurazioni di durata e consumo per ogni algoritmo. Tutti i comandi software necessari sono stati eseguiti su Raspberry Pi 2, remotamente via SSH da un computer portatile. Si eseguono quindi i seguenti passi:

1. Codifica dell'algoritmo di ordinamento in linguaggio C. L'accorgimento principale riguarda la scrittura in dimensione contenuta e il più possibile efficiente.

2. Caricamento e compilazione dell'algoritmo su Raspberry Pi 2 tramite il software GCC (versione 5.2 [21]) con flag -O3 che promette la massima ottimizzazione post-compilazione possibile.

3. A questo punto si esegue nella shell il comando:
   ```
   # ulimit -s 262144
   ```
   Questo aumenta la dimensione dello stack associato ad ogni processo eseguito all'interno di quella shell. Più precisamente la dimensione associata sarà 262144 KByte.
   Il comando permetterà agli algoritmi di manipolare array fino a 60 milioni di elementi senza incorrere in errori dovuti al superamento della dimensione dello stack.

4. Possiamo ora effettuare le misurazioni temporali ed energetiche.
   Un esempio di comando sarà quindi:

```
# time ./quicksort.x 10000
```

Questo sfrutta il comando **time** per ottenere la misura in millisecondi del tempo di esecuzione dell'algoritmo compilato. Nell'esempio stiamo eseguendo il quicksort passandogli come parametro la dimensione del vettore da ordinare, in questo caso 10000 elementi. Tutti gli algoritmi hanno al loro interno una funzione che genera un vettore di numeri casuali di dimensioni pari al parametro passatogli.
L'algoritmo dovrà quindi ordinarlo e terminare.
Durante l'esecuzione dell'algoritmo si registra in un video il display dell'amperometro di precisione, ovvero si registrerà l'assorbimento in mA alternati istante per istante visualizzato dall'amperometro di precisione.

5. Una volta terminata l'esecuzione avremmo ottenuto due dati:

   (a) Il primo riguarda la durata totale dell'esecuzione in millisecondi fornita dal comando **time**.

   (b) Il secondo è il video che ha registrato le variazioni nell'assorbimento di corrente di Raspberry Pi 2 durante l'esecuzione dell'algoritmo. Da questo si otterrà il valore dell'assorbimento medio durante l'esecuzione, annotando ogni secondo il valore visualizzato e effettuando una media su tutti i valori ottenuti.
   Moltiplicando l'assorbimento medio della corrente alternata per la tensione di rete (i 220 Volt), si ottiene il valore in Watt del consumo medio de dispositivo. Considerando quindi il consumo medio in Watt possiamo ipotizzare che se l'algoritmo fosse eseguito sulla stessa dimensione dell' array ripetutamente per un'ora, abbiamo il consumo medio in Wattora per quell'algoritmo.

   Potremmo ora annotare tutte le informazioni necessarie per le eventuali comparazioni:

   - Il nome dell'algoritmo testato
   - La dimensione dell'array che è stato ordinato passata come parametro
   - Il tempo totale di esecuzione dell'algoritmo
   - Il consumo medio in Wh(Wattora) dell'algoritmo

Iterando questa procedura per tutti gli algoritmi si ottiene l'insieme tutti i dati di interesse riferiti agli algoritmi e alla dimensione dell'array testata.

Per rendere più chiari i grafici e poter comprendere più chiaramente il significato delle differenze energetiche, i consumi saranno comparati al tempo di scaricamento di una batteria.

## 1.5 Rappresentazione e comparazione dei risultati

I risultati ottenuti alla fine della procedura di test forniscono un tempo di esecuzione unito ad un consumo in Wh medio della Raspberry Pi 2.

Da queste misure si possono ottenere, grazie ad una rielaborazione, due dati di interesse:

1. Ipotizzando di alimentare la Raspberry Pi 2 con una **batteria di riferimento da 5 Volt e 1000 mAh**, si può stimare per ogni algoritmo in quanto tempo si porti la batteria allo scaricamento completo se è fatto eseguire ripetutamente su un array della stessa dimensione.

2. É possibile stimare il numero totale di elementi che l'algoritmo, in esecuzione ripetuta sul dispositivo collegato alla batteria di riferimento, riesce ad ordinare prima del completo scaricamento della stessa.
   Questo concetto parlerà di *efficienza* dell'algoritmo, considerando come efficienza il numero totale di array che sono stati ordinati prima del completo scaricamento della stessa. Per alcuni algoritmi saremo in grado di stimare esattamente il numero di elementi ordinati (vedi Capitolo 4).

Per ogni serie di algoritmi avremmo i due grafici sopra descritti che ci daranno una panoramica più precisa dei risultati ottenuti.

É stato scelto di rielaborare i dati in riferimento ad una batteria, per rendere i risultati allineati con le problematiche reali dei dispositivi mobili che spesso ne sono equipaggiati.

Il dato di interesse dovrebbe aiutare a capire quanto un particolare processo software incida sulla durata complessiva della batteria, tenendo anche in considerazione la quantità totale di lavoro che riesce a svolgere prima del suo completo scaricamento. Su questi dati saranno proposte alcune considerazioni che aiuteranno a delineare l'importanza o la neutralità di alcuni risultati.

# Capitolo 2

# Aree di ricerca

## 2.1 Articoli di riferimento

Per contestualizzare il lavoro di ricerca e studio dei risultati che andremo a svolgere, per prima cosa ci soffermeremo sugli articoli di riferimento che possono essere di aiuto nel chiarificare metodi di test e gli obbiettivi della ricerca.
Il principale documento di riferimento in questo ambito, che ha ispirato un altro articolo sullo stesso argomento [8], è il *"Exploring the Energy Consumption of Data Sorting Algorithms in Embedded and Mobile Environments"* [9] redatto nel 2009 da quattro ricercatori tedeschi, con lo scopo di indagare l'effetto degli algoritmi di ordinamento su una architettura embedded.

Sono stati analizzati molti algoritmi di ordinamento standard, tra i quali troviamo: Quicksort, Mergesort, Insertionsort e Heapsort. Per ogni algoritmo sono stati misurati i consumi e il tempo di esecuzione in millisecondi, infine i dati sono stati memorizzati all'intero di un database. Le misurazioni ottenute sono poi state analizzate per ottenere come risultati dei grafici che cerchino di identificare più chiaramente il trend dei consumi energetici, per ogni algoritmo di ordinamento, all'aumentare del numero di elementi da ordinare.

### 2.1.1 Il Testbed utilizzato

Il Testbed progettato in questo caso specifico è composto di più strumenti avanzati che dovrebbero garantire la più alta affidabilità nei risultati energetici, riducendo eventuali errori di misura o componenti parassite che potrebbero influenzare le misurazioni. Inoltre si è cercato di ottenere misurazioni che siano riferite solamente al microcontrollore testato, non inserendo nella misurazione tutti i componenti di supporto al microcontrollore durante l'esecuzione degli algoritmi.

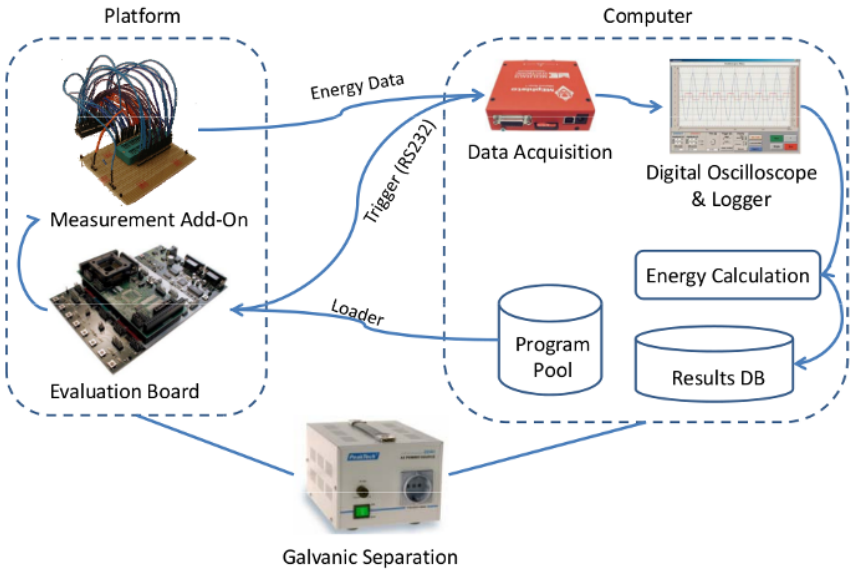


Figura 2.1: Testbed del documento di riferimento [9]

La scelta dell'utilizzo di microcontrollori, in questo caso gli ATMega 16, 32 e 128 [4], è è motivata come segue:

- La mancata presenza di un sistema operativo per l'esecuzione degli algoritmi che sarebbe potuto diventare influente sul tempo di esecuzione e sul consumo energetico speso dagli algoritmi testati. GLi algoritmi sono stati quindi compilati, caricati nella memoria del microcontrollore ed eseguiti uno per volta.

- La presenza di una testboard per questi microcontrollori adatta alla programmazione e alle misurazioni quali quelle relative all'assorbimento e ai consumi energetici [5].

- Un ampio utilizzo di questa tipologia di microcontrollori nei dispositivi embedded. L'ambito attualmente di maggior riferimento per questo tipo di dispositivi è quello elettronico e della automazione industriale.

### 2.1.2 Risultati ottenuti

I risultati dell'esecuzione degli algoritmi di ordinamento sui vari microcontrollori sono rappresentati nei grafici seguenti:

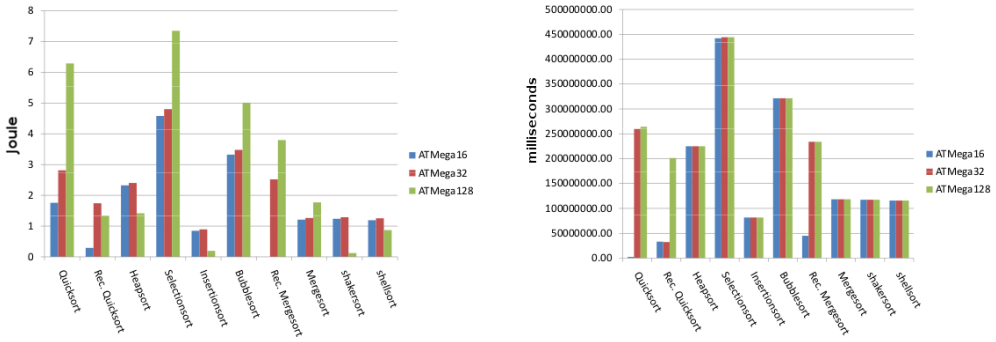


Figura 2.2: Risultati riguardati il tempo di esecuzione e il consumo energetico in Joule, degli algoritmi di ordinamento sui tre tipi di microcontrollore.

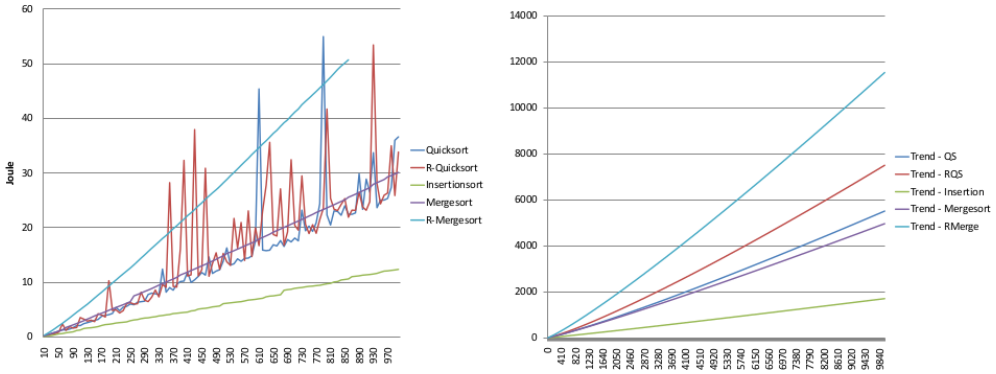


Figura 2.3: Risultati riguardati i trend di consumo energetico al crescere del numero di elementi dei vari algoritmi di ordinamento.

Dai grafici è possibile notare come l'algoritmo che mostrerebbe avere un trend energetico il più efficiente possibile sia l'InsertionSort. Questo algoritmo al contrario di controparti note come MergeSort e Quicksort, ha una complessità quadratica diventando efficiente solamente in presenza di array quasi ordinati. Risulta quindi molto interessante che InsertionSort seppur non sia dotato di una complessità ottimale, come ad esempio l'algoritmo MergeSort avente complessità $O(Nlog_2N)$, risulterebbe, per dimensioni di array crescenti, il più efficiente tra gli algoritmi testati.

Inoltre si potrebbe notare che le implementazioni ricorsive di alcuni algoritmi (Quicksort e Mergesort) risulterebbero avere una efficienza energetica peggiore della loro controparte iterativa. Difatti mentre il MergeSort iterativo si presenta come il secondo algoritmo per efficienza, la sua controparte ricorsiva risulterebbe peggiore rispetto a tutte le altre implementazioni algoritmiche testate. Le implementazioni ricorsive hanno lo svantaggio di utilizzare memo-

ria di sistema per il tracciamento delle chiamate ricorsive e i loro dati. Questo tipo di risultati sarebbero allineati con il quantitativo di memoria ausiliaria disponibile durante l'esecuzione, che per questi tipi di microcontrollori, come da specifiche, è molto limitata.

### 2.1.3 Differenze tra le architetture

I risultati ottenuti sulle architetture a microcontrollore suscitano un particolare interesse in quanto mostrerebbero di non essere allineati con delle possibili previsioni basate sulla complessità degli algoritmi testati. In questo caso un algoritmo quadratico risulterebbe più efficiente rispetto ad un algoritmo avente complessità inferiore.

É incoraggiante, alla luce di queste considerazioni, valutare di applicare questa tipologia di test energetico anche ad architetture differenti dai microcontrollori, che siano sempre di interesse per l'ambito embedded. Nel nostro caso è stato scelto l'ambito embedded dei dispositivi mobili, che vorrebbe rappresentare l'insieme dei cellulari, tablet, phablet e smartwach, che stanno sempre più spopolando nel mercato tecnologico mondiale [31].

L'architettura attualmente di riferimento nel marcato mobile è l'architettura ARM [2], i cui vantaggi principali consistono in un basso consumo energetico unito ad una buona capacità di calcolo, soprattutto in ambito multimediale. Per ottenere le prestazioni raggiunte, nelle varie versioni dei processori ARM, sono state aggiunte delle ottimizzazioni hardware unito ad un incremento della quantità e velocità della memoria di sistema.

La Raspberri Pi 2 che utilizzeremo nei nostri test è equipaggiata con un SoC (System-on-Chip) Broadcom BCM2836 [7] dotato di un processore ARM Coretex-A7 [3]. Questa tipologia di processore quad-core dotato di cache di primo e secondo livello, è utilizzato in molti altri dispositivi mobili, soprattutto negli smartphone. Accompagnato sulla Raspberry Pi 2 da una 1 GByte di RAM, dovrebbe essere rappresentativo per la maggior parte dei dispositivi cellulari, tablet e mobile presenti sul mercato.

I nostri test, alla luce di quelli eseguiti sui microcontrollori Atmel, cercheranno di poter dare una visione d'insieme del comportamento energetico degli algoritmi di ordinamento su un'altra architettura embedded di riferimento, l'architettura ARM.

## 2.2 Considerazioni utili e ambiti di ricerca

Per arrivare ad una diminuzione dei consumi nell'esecuzione di un algoritmo, si sono ricercati dei metodi di approccio che possano successivamente 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 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 sulla diminuzione della complessità nel caso pessimo e nel caso migliore. Dovremmo ottenere in tal modo un miglioramento nel caso medio, compreso tra la massima e la minima complessità, che potrebbe dare risultati interessanti per quanto riguarda l'ambito dei consumi.

2. ***Numero totale di scritture:***
   Una caratteristica che potrebbe essere cruciale dal punto di vista dei consumi energetici è il numero totale 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 due volte quello associato alle letture [23, 39]. 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 [11].

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 vantaggiosa 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 abbastanza interessanti, possiamo anche pensare ad un aspetto ulteriore. La Cache come tipo di memoria è organizzata su due o tre livelli [10]. 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 [32], 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 possa essere, per quanto riguarda il codice da eseguire e 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 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.
   L'operazione di aggiunta di ulteriore memoria a tempo di esecuzione, comporterà un utilizzo della stessa, che si trasformerà in un numero di

letture e scritture arbitrario. Si deve inoltre considerare 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 [22, 29]. 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 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.
   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.
   Due ulteriori aspetti, di riflesso, potrebbero incidere 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 sui comandi a seconda delle regole sintattiche imposte dal linguaggio.
   Il secondo si riferisce alla dimensione in memoria di una struttura dati complessa, che potrebbe risultare diverse 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 gli aspetti. La politica adottata sarà quindi quella di utilizzare strutture dati il più semplici possibili. Sarà perciò utilizzata come struttura dati *l'array di Interi*.

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

1. ***Standard swap [40] vs Xor swap [44]:***
   All'interno degli algoritmi di ordinamento una funzione sicuramente

importante è sicuramente la funzione che effettua gli scambi. Il metodo generalmente utilizzato è quello in cui si utilizza una variabile temporanea di supporto che permette la transizione temporanea di un valore che sarà riscritto successivamente nella sua posizione finale. Questo metodo prevede l'utilizzo 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 interessante 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.
A riguardo si è scelto di studiare lo Xor swap, una funzione di scambio che sfrutta le proprietà della funzione di Bitwise Xor [6] per scambiare due variabili senza aver bisogno di una terza variabile di appoggio.

2. ***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 [28] 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.

3. ***Min comparison sort [27] ed algoritmi di ordinamento non uniformi integrati con algoritmi standard:***
   Nel terzo volume del libro *"The Art of Computer Programming"* [27] nella sezione 5.3, si fa riferimento alla problematica del Min comparison

sort. Questo ambito di studio indaga le possibilità e i metodi algoritmici per ordinare un gruppo di elementi effettuando il minor numero possibile di confronti. I confronti necessari per ordinare un gruppo di elementi possono essere visti sotto la forma di un albero binario. Minimizzare il numero di confronti consiste nel minimizzare il percorso più lungo tra la radice ed una foglia dell'albero, ovvero ridurre per ogni permutazione il numero di confronti necessari per identificarla e quindi poterla ordinare. Un approccio che aiuta a raggiungere questo risultato consiste nell'utilizzo di algoritmi non uniformi. Ad ogni dimensione della permutazione in ingresso, è associato un algoritmo di ordinamento scritto e ottimizzato nel numero di confronti, appositamente per quella dimensione.
Gli algoritmi non uniformi che cercheranno di effettuare il minor numero di confronti, basandosi su una albero binario di confronti il quale cresce esponenzialmente rispetto alla dimensione della permutazione da ordinare, saranno realizzati per valori piccoli della dimensione dell'input così da poter avere un codice facile da correggere e utilizzare.
Inoltre questi algoritmi saranno inseriti in un altro algoritmo di ordinamento, il Quicksort, al fine di ridurre la sua complessità ma soprattutto per studiare come l'utilizzo di algoritmi ottimi nei confronti aiuti a ridurre il consumo energetico complessivo dell'ordinamento. Le motivazioni dettagliate del perché sia stato utilizzato Quicksort come algoritmo per l'integrazione degli algoritmi non uniformi ai fini dei test energetici, sono descritte in dettaglio nella sezione 5.4.1.

# Capitolo 3

# Standard swap e Xor swap

## 3.1 Panoramica generale

La prima analisi sulla quali ci siamo soffermati riguarda l'implementazione degli scambi in una procedura di sorting. Questa problematica è primaria nell'esecuzione della maggior parte degli algoritmi di ordinamento. Trattandosi di una operazione fondamentale, una sua ottimizzazione rispetto al consumo energetico determinerebbe immediatamente una miglioria analoga per l'algoritmo che ne fa 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 legati ad operazioni da effettuare tra le variabili da scambiare. Possiamo citarne un paio:

- Scambio tramite somma e sottrazione tra interi [41]
- Scambio tramite Xor [44]

Si è deciso di studiare solamente lo scambio tramite Xor, 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 generalmente più complesse da effettuare che una operazione di Bitwise Xor richiesta dal metodo scelto. Un esempio di scambio tramite Bitwise Xor si può vedere applicato nell'implementazione 3.2.

## 3.2 Implementazioni utilizzate

L'algoritmo utilizzato per testare lo scambio mediante Xor è molto semplice. All'interno di un ciclo for andremo a scambiare tra loro due variabili, utilizzando nel primo caso lo Standard swap [40] 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.
Il codice risultante 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 proprieta 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 esplicativi. Il metodo dello Standard swap sembra mostrare risultati migliori rispetto allo Xor swap. Non solo si mostra nettamente più veloce ma mostra anche un consumo nettamente inferiore. I grafici relativi sono riportati nelle figure 3.1 e 3.2 .

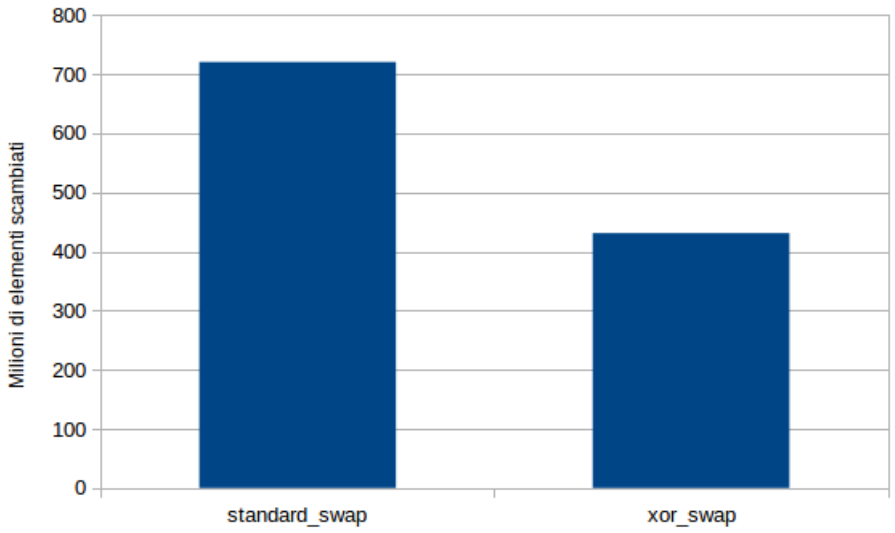


Figura 3.1: Scambi effettuati prima dello scaricamento completo della batteria di riferimento.

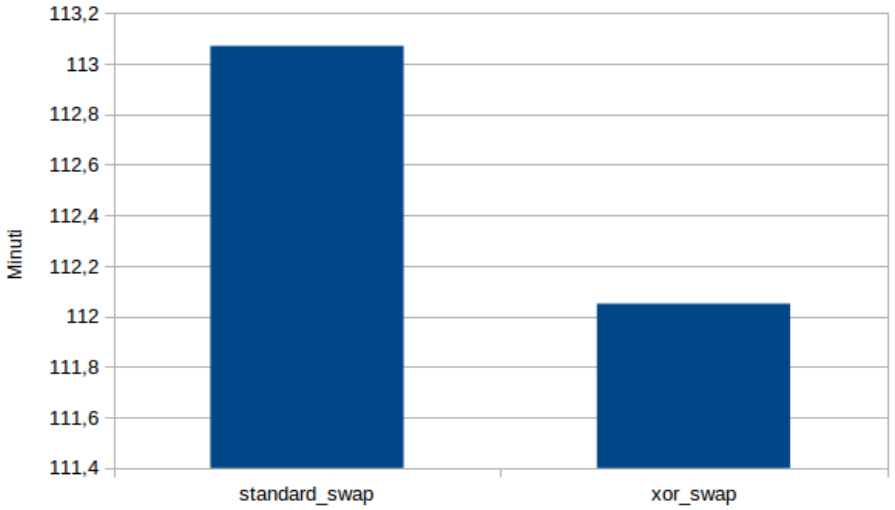


Figura 3.2: Durata della batteria di riferimento.

Tali risultati sperimentali potrebbero avere la seguente spiegazione. Più precisamente:

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

$$1\,allocazione\,di\,variabile\,intera + 3\,assegnamenti$$

Andando ad analizzare le ottimizzazioni effettuabili dal compilatore si scopre che 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 [38]. Perciò il consumo significativo avviene solamente nel momento degli assegnamenti.

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

$$3\,assegnamenti + 3\,operazioni\,di\,BitwiseXor\,tra\,interi$$

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

Studiando la differenza tra le due funzioni notiamo fondamentalmente un eccesso nella seconda di 3 operazioni di Bitwise Xor non banali. É 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 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 cambio efficiente. A fronte della situazione reale possiamo affermare che in generale 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 come proposta di miglioramento del primo, che promettono il minor numero di scritture in assoluto:

1. ***CycleSort*** *[24]*
2. ***OlivatoCycleSort***

Questi due algoritmi hanno i 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 è esattamente $\Theta(N^2)$ [15], ovvero la complessità nel caso migliore è uguale al caso peggiore ed è quadratica, mentre il secondo algoritmo, che si propone come variante ottimizzata del primo, ha una complessità compresa tra $\Omega(N)$ nel caso migliore e $O(N^2)$ nel caso peggiore.

Queste differenze nella complessità andranno ad incidere sui risultati temporali ed energetici. Ricordiamoci che i confronti sono abbastanza onerosi

sia energeticamente che da un punto di vista computazionale e che quindi un valore quadratico non può che essere preponderante confrontato con la linearità delle scritture. Ma andiamo comunque a descriverli in dettaglio.

## 4.2 Il CycleSort

Il CycleSort [24] 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. Per comprenderli più in dettaglio capiremo come ogni singolo elemento sarà quindi ordinato e come questa proprietà potrà essere sfruttata iterativamente per ordinare un array di interi.

#### 4.2.1.1 Ordinare un elemento

La proprietà basilare che deve valere per ogni elemento è la proprietà che assicura il riposizionamento finale di ogni elemento affinché rispetti la caratteristica di ordinamento desiderata. La proprietà di ordinamento scelta da questo algoritmo mira a ordinare ogni elemento (e quindi scriverlo nella posizione ordinata) in un solo passaggio. Per fare ciò l'elemento da ordinare è confrontato con tutti gli altri elementi ed è memorizzato il numero totale di elementi più piccoli dell'elemento scelto.
Il dato ottenuto indicherà quale debba essere la posizione finale dell'elemento affinché esso possa considerarsi ordinato. Se l'ordinamento desiderato è crescente e l'array è indirizzato con indici crescenti da sinistra verso destra, allora il numero di elementi inferiori rappresenta esattamente la posizione finale dell'elemento, altrimenti se l'ordinamento è decrescente la posizione in cui dovrà essere collocato si troverà sottraendo questo valore al numero totale di elementi.

#### 4.2.1.2 Il ciclo di ordinamento

Preso un elemento A da ordinare e trovata la sua posizione finale tramite la proprietà di ordinamento scelta, se lo si vuole collocare nella posizione trovata

ci si accorge che questa è già occupata da un altro elemento B dell'array. L'elemento B, se diverso da A, occupando la posizione finale di un altro elemento risulta non essere ordinato globalmente. Iterando sull'elemento B la proprietà di ordinamento utilizzata per l'elemento A, in generale, potrebbe essere trovata come posizione finale la posizione occupata da un altro elemento C.
Continuando a iterare su ogni elemento trovato che occupa una posizione non corretta all'interno dell'array, si arriverà a trovare quell'elemento che ha come posizione finale la posizione occupata originariamente dall'elemento A. Quest'ultimo elemento a chiuso un ciclo di ordinamento, ovvero era l'ultimo elemento trovato ciclicamente che non era nella posizione corretta, scoperto iniziando con l'ordinamento di A e arrivando ad essere riposizionato nella posizione lasciata "vuota" proprio dall'elemento A di partenza.
A livello teorico questa spiegazione può essere formalizzata come una permutazione ciclica di elementi [16], finalizzata all'ordinamento dell'array.

#### 4.2.1.3 Ordinare un array

Un ciclo di ordinamento è quindi in grado di ordinare, posizionando nella loro posizione finale (quella dell'array già ordinato), gli elementi che incontra durante il riposizionamento ciclico. Se un ciclo di ordinamento riposiziona esattamente un numero di elementi pari al numero totale di elementi presenti nell'array, è facile intuire come essendo ogni elemento dell'array riposizionato nella sua posizione finale, tutto l'array risulti ordinato. Generalmente non è detto che un ciclo di ordinamento riesca ad ordinare tutti gli elementi dell'array. Normalmente è in grado di ordinarne un suo sottoinsieme.

Il CycleSort sfrutta questa proprietà cercando di applicare per ogni elemento dell'array partendo da sinistra un ciclo di ordinamento. Se l'elemento è già ordinato non lo riscrive e passa al successivo, altrimenti il ciclo di ordinamento riposizionerà un gruppo di elementi. Cercando, ed eventualmente eseguendo, un ciclo di ordinamento su ogni elemento dell'array, il CycleSort si assicura che la posizione di finale di ogni elemento sia corretta, ottenendo in definitiva tutti gli elementi dell'array ordinati e quindi l'array ordinato.

### 4.2.2 Complessità e numero totale di scritture

Per quanto riguarda la complessità abbiamo esattamente $\Theta(N^2)$ [15]. Infatti per ogni elemento, partendo dal primo elemento a sinistra, dovendo confrontarlo con tutti gli altri elementi nell'array per trovare la sua esatta posizione effettueremo in tutto $N-1$ confronti, con N il numero di elementi presenti nell'array.

Essendo N il numero di elementi da ordinare otteniamo il numero totale di confronti come: $N * (N - 1)$ approssimato con $N^2$.
Lo svantaggio principale di questo approccio consiste nel fatto che una volta terminato un ciclo di ordinamento, l'algoritmo non è in grado di riconoscere quali elementi a sinistra della sua posizione siano già stati ordinati da un precedente ciclo di ordinamento.

Per quanto riguarda le 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 per ordinare l'array ogni elemento dovrà essere riposizionato e perciò 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 dovro' 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 e' gia' 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 piu' 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

Questa proposta di algoritmo 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 soprattutto la complessità minima dell'algoritmo.

### 4.3.1 Struttura e idea dell'algoritmo

Questa variante del CycleSort cerca di proporre un miglioramento della complessità minima.
Come base per ottenere il minor numero di scritture utilizza anch'esso i cicli di ordinamento. La differenza principale consisterà in come sarà effetuata la ricerca dei cicli all'interno dell'array.

#### 4.3.1.1 La proprietà transitiva dell'ordinamento

In un ordinamento tra elementi deve valere la proprietà transitiva. Se un elemento A è ordinato rispetto ad un elemento B e l'elemento B è ordinato rispetto ad un elemento C, allora l'elemento A sarà ordinato anche rispetto all'elemento C. Questa proprietà ci permette di scoprire se un array di elementi è ordinato globalmente semplicemente controllando che ogni elemento sia ordinato rispetto al suo successore. Inoltre è utilizzata da un algoritmo di ordinamento classico, l'InsertionSort [25], per trovare elementi non ordinati e riposizionarli in ordine rispetto al sottoarray di sinistra.

La complessità minima risulta essere N confronti, in quanto se l'array è già ordinato l'algoritmo non farà altro che verificare l'ordinamento tramite la proprietà transitiva e terminare. Si è cercato di integrare questa proprietà nel CycleSort con l'intento di trovare rapidamente i cicli di ordinamento da dover eseguire e ridurre la complessità dell'algoritmo nel caso in cui l'array sia già ordinato o quasi ordinato. Nel nostro caso l'integrazione di questa proprietà non porterà ad una variazione del numero di scritture ma solamente ad una riduzione della complessità media dell'algoritmo.

#### 4.3.1.2 I passi di esecuzione dell'algoritmo

Per chiarire come l'algoritmo coniughi i cicli di ordinamento con la ricerca tramite la proprietà transitiva, di seguito sono descritti i passi effettuati. Il primo passo, consiste nello scorrere l'array partendo da sinistra verso destra, assumendo che gli indici dell'array siano crescenti da sinistra verso destra, fino a che gli elementi tra loro soddisfino la proprietà transitiva dell'ordinamento con cui si è scelto di ordinare. Se durante lo scorrimento troviamo un elemento che non soddisfa questa proprietà, abbiamo trovato un elemento minore del massimo elemento nel sottoarray di sinistra.

Questa condizione ci dice che è possibile che l'elemento appena trovato non sia nella sua posizione corretta, in riferimento all'ordinamento globale, e una certezza sul fatto che tutti gli elementi maggiori dell'elemento trovato contenuti nel sottoarray di sinistra, non siano ordinati rispetto all'ordine globale. Possiamo dedurre inoltre che se una parte degli elementi di sinistra

non è ordinata, implica che esisterà nella parte destra uno o più elementi non ordinati, elementi di un ciclo di ordinamento non ancora effettuato che li andrà ad ordinare.

Il secondo passo consiste nell'andare ad eseguire un ciclo di ordinamento scegliendo l'elemento appena trovato, esterno al sottoarray di sinistra. 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) Questo ciclo più quelli passati hanno ordinato globalmente N elementi e quindi sono nel caso un cui il vettore è ordinato e termino.

   (2) Questo 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 comprendiamo che se non si sono ordinati tutti gli elementi del mio array con questo ciclo potrebbe esistere almeno un elemento non ordinato globalmente.
   Distinguiamo anche qui due casi:

   (1) Se ripartendo con un ciclo che verifica la proprietà transitiva, proprio dalla posizione appena ordinata globalmente, non troviamo nessun elemento che non soddisfi la proprietà, questo implica per transitività che la porzione a destra dell'elemento è ordinata rispetto a quest'ultimo. Ma dato che siamo partiti da un elemento che era stato precedentemente ordinato globalmente se tutti gli elementi successivi sono ordinati rispetto ad esso allora sono obbligatoriamente ordinati anche globalmente. Avendo così ordinato l'array, termino.

   (2) Se ripartendo con un ciclo che verifica la transitività tra gli elementi 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 all'inizio del secondo passo.

II. Nel secondo caso l'elemento non ha trovato un ciclo da eseguire. Sicuramente è già posizionato correttamente rispetto all'ordinamento globale. Ciò implica che oltre agli elementi minori di se stesso contenuti nel sottoarray ordinato di sinistra esistono esattamente M elementi minori di se stesso a destra, con M il numero di elementi maggiori di se stesso

contenuti nel sottoarray di sinistra.
Alla luce di queste deduzioni 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 destra. L'elemento nuovamente trovato è un nuovo possibile candidato a poter iniziare un nuovo ciclo di ordinamento, tornando ricorsivamente all'inizio di questo passo di ordinamento.

### 4.3.2 Complessità e numero totale di scritture

La complessità di questo algoritmo è $O(N^2)$. Possiamo però essere più precisi provando 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. Infatti il ciclo di di verifica della proprietà transitiva noterà che partendo dall'inizio dell'array è arrivato alla fine senza aver effettuato cicli di ordinamento, ma solamente confrontato transitivamente gli elementi.

Il caso peggiore si presenta se si considera un vettore ordinato e si applica uno shift a destra degli elementi di una posizione, ricollocando l'elemento massimo, che uscirebbe dall'array, in prima posizione. In questo caso il primo ciclo di verifica della transitività 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 e per farlo per ogni elemento dovrà essere confrontato con tutti gli altri $N-1$ elementi dell'array. Essendo N il numero totale degli elementi dell'array si calcolerà il numero totale di confronti come $N*(N-1)$ approssimato ad $N^2$. Una volta terminato questo ciclo il contatore noterà che ha ordinato esattamente N elementi e perciò terminerà.

Quello che ad una analisi più approfondita si può notare è che il caso medio di complessità di questo algoritmo è 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, come visto in un esempio precedente, al massimo N scritture.

```
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 dovro' 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 piu' 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 finche' c'e' 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 una maggior incidenza sul carico energetico e computazionale del numero quadratico di confronti rispetto al numero lineare di scritture. Nonostante questo si possono notare delle differenze tra i due algoritmi, che vedono l'OlivatoCycleSort presentare risultati migliori sia dal punto di vista dell'efficienza energetica, sia dal punto di vista della durata di vita della nostra batteria di riferimento. I grafici mostrano solamente una comparazione tra questi due algoritmi in quanto una comparazione più ampia con algoritmi classici farebbe notare come entrambi siano inefficienti energeticamente e temporalmente a causa dell'influenza importante della complessità quadratica. Nei grafici stimeremo il numero di singoli elementi

ordinati in ogni array, considerando che ogni ciclo di ordinamento riposiziona ogni elemento nella posizione finale ordinata globalmente, quindi una volta posizionato lo considereremo già ordinato.

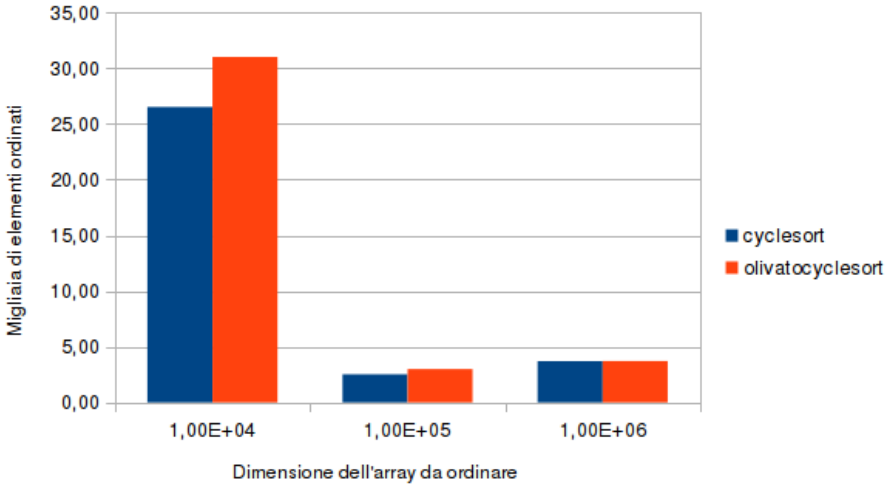


Figura 4.1: Numero totale di elementi ordinati prima del completo scaricamento della batteria di riferimento utilizzando tre differenti lunghezze di vettore.

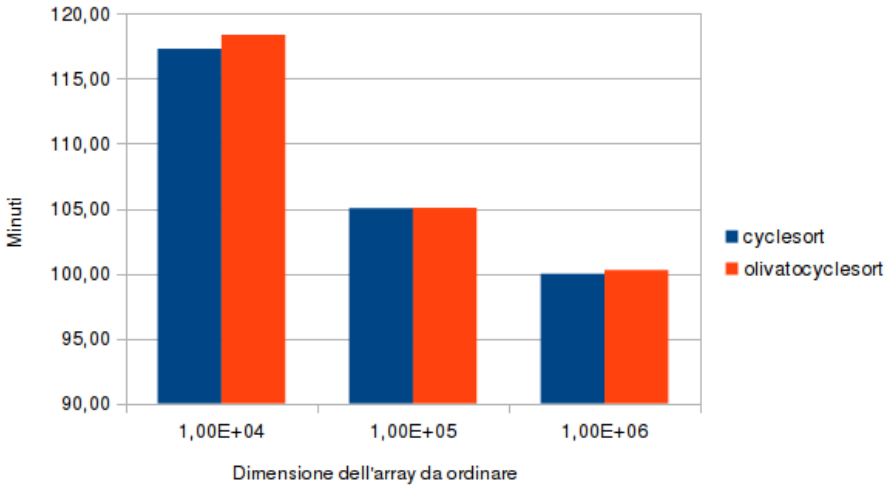


Figura 4.2: Durata della batteria di riferimento rispetto alle tre lunghezze di vettore.

## 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 [28], il costo energetico totale risulterebbe solamente legato al numero di scritture. Si possono notare già i primi esempi di architetture orientate a questa particolare implementazione all'interno dei progetti di calcolatori quantistici [12] 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 di interesse, 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. É chiaro che in questo contesto la vita del supporto si allungherebbe 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) [43]. 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 [34], 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 senso, 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 interessante risulta quella dell'approccio non uniforme. Ovvero ordinare gli elementi seguendo un albero di decisione che tenga traccia della permutazione in ogni suo nodo e dove la foglia contenga l'esatto numero minimo di scambi necessario ad ordinare quella permutazione. Questo approccio è in grado di minimizzare i confronti necessari ad ordinare una permutazione diventando una possibile soluzione del problema del Minimum Comparison Sort, ovvero riuscire ad ordinare un gruppo di elementi con il minor numero assoluto di confronti. Essendo i confronti rilevanti sul bilancio energetico complessivo di un algoritmo di ordinamento una loro minimizzazione potrebbe portare ad una riduzione nell'ambito dei consumi.

## 5.2 Minimum Comparison Sorting

Il numero minimo di confronti necessari ad ordinare N elementi, in generale è pari a *zero*. Esistono infatti altri tipi di algoritmi capaci di ordinare un gruppo di elementi senza effettuare confronti tra essi [14, 35]. Questi algoritmi utilizzano particolari proprietà dell'insieme da ordinare per non dover confrontare due elementi tra loro.

Nel caso in cui si voglia indagare il minor numero di confronti in un algoritmo basato su confronti, bisogna prima rappresentare più chiaramente il caso di studio. Un metodo di ordinamento che utilizzi i confronti può essere visto in generale come un albero binario. Ogni nodo dell'albero, partendo dalla radice, rappresenta un confronto tra due elementi da ordinare. Il ramo destro collegato a quel nodo rappresenterà la condizione in cui il primo elemento sia maggiore del secondo, mentre il ramo sinistro rappresenterà la condizione

in cui il primo elemento sia minore o uguale del secondo. Si crea così un percorso tra la radice e la foglia, dove la foglia rappresenta la permutazione identificata da quella serie di confronti.

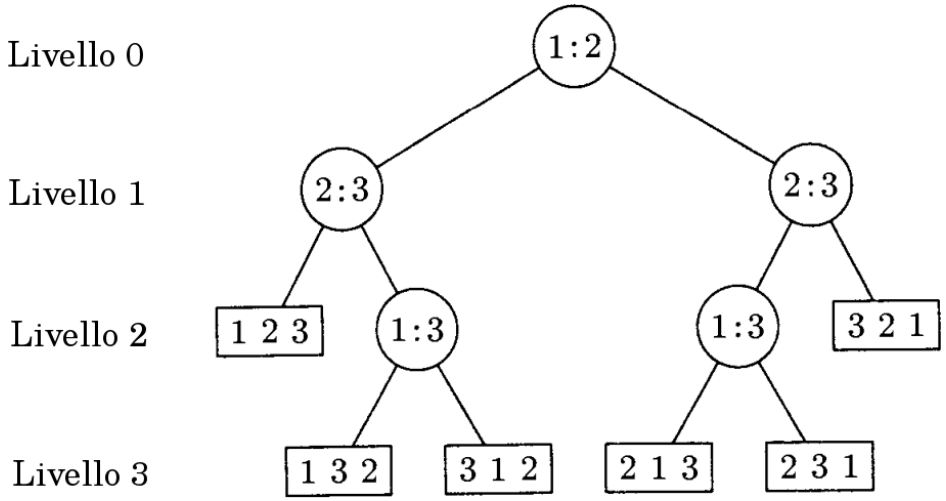


Figura 5.1: Un albero di confronti per l'ordinamento di tre elementi.

Per ricercare il minor numero di confronti dobbiamo assicurarci che non avvengano confronti ridondanti, ovvero che due elementi non siano confrontati due volte tra loro, all'interno di un percorso radice-foglia. Per la proprietà transitiva della relazione di ordinamento, i confronti ridondanti risultano essere anche quei confronti che non tengano conto della transitività dei confronti effettuata in precedenza. Tutte le permutazioni di un gruppo di elementi di ingresso sono possibili, e ogni permutazione definisce un percorso univoco dalla radice ad un nodo esterno (foglia); da questo segue che ci sono esattamente $N!$ nodi esterni in un albero di confronti che ordina N elementi senza confronti ridondanti.

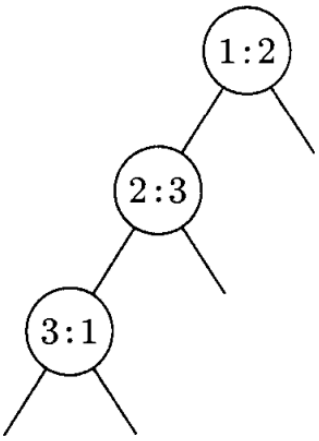


Figura 5.2: Un esempio di confronto ridondante tra l'elemento 1 e 3.

Consideriamo $S(N)$ come il minimo numero di confronti sufficienti ad ordinare N elementi. Se tutti i nodi interni dell'albero di confronti sono nei livelli inferiori a $k$, allora, per la natura ad albero binario dei confronti, ci possono essere al massimo $2^k$ foglie dell'albero.
Imponendo che $k = S(N)$, otteniamo

$$N! \leq 2^{S(N)}$$

dove $N!$ è il numero effettivo delle foglie presenti nel nostro albero binario. Essendo $S(N)$ un intero e avendo ipotizzato di ordinare un insieme non vuoto di elementi ($N \neq 0$), possiamo riscrivere la formula ottenendo il valore minorante

$$S(N) \geq \lceil log_2(N!) \rceil$$

Dall'approssimazione di Stirling [1] possiamo dire che

$$\lceil log_2(N!) \rceil = Nlog_2N - N/log_22 + \frac{1}{2}log_2N + O(1)$$

quindi sono necessari circa $Nlog_2N$ confronti. Questo valore è legato al caso peggiore, infatti in un albero di confronti ottimizzato il percorso minimo riscontrabile tra la radice ed una foglia è costituito da $N-1$ confronti. Questo valore è confermato dal fatto che in un ipotetico caso in cui il vettore si già ordinato, confrontare ogni elemento con il successivo verificando che la proprietà di ordinamento sia rispettata nell'ordine voluto (da destra a sinistra o da sinistra verso destra), è sufficiente ad ottenere il vettore ordinato.

Nel nostro caso il raggiungimento del minor numero di confronti, potrebbe avere un impatto sui consumi degli algoritmi e ridurre contemporaneamente la complessità, riducendo il tempo di esecuzione dell'algoritmo. Per ottenere

un albero di confronti ottimo dovremmo generarlo e specializzarlo per una particolare dimensione dell'insieme di elementi da ordinare, sfruttando quindi un approccio non uniforme.

Per ulteriori approfondimenti sul problema del Minimum Comparison Soting si consiglia il terzo volume del libro *"The Art of Computer Programming"* [27] nella sezione 5.3, *Optimum Sorting*.

## 5.3 Ordinamento non uniforme

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 dimensione della combinazione in ingresso.

2. Ogni combinazione sarà una foglia dell'albero e la complessità per raggiungerla sarà la lunghezza del percorso dalla radice alla foglia. L'idea è rendere minimo il percorso tra la radice e la foglia che è l'identificativo di come ordinare quella permutazione.

3. Una volta identificata la singola combinazione sarebbe anche possibile ottimizzare il numero degli scambi e quindi della scritture per la specifica combinazione, rendendo minimo il numero totale 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 dimensione dell'input. Purtroppo non è praticamente 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 ottenere un algoritmo non uniforme che ordini tramite un albero di confronti un array di 15 elementi. Al suo interno avrà del codice per l'ordinamento di ogni combinazione che risiederà nelle foglie dell'albero binario. Calcoliamo solamente il numero totale delle foglie, ovvero il numero delle possibili combinazioni di 15 elementi. La formula per il calcolo delle possibili combinazioni di N elementi è $N!$. Avremmo quindi 15! foglie dell'albero, con il loro specifico codice atto ad ordinarle.

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, 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 restituirebbe:

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

Il vero risultato risulterebbe molto maggiore di quello ottenuto e quindi per un sistema dover eseguire un codice che pesi più di 130 GByte crea diversi problemi, sia dal punto di vista della memoria centrale, sia dal punto di vista della località del codice. L'albero dei confronti, per sua natura, avrà dei salti tra punti di codice anche distanti tra loro, il che comporterebbe un continuo processo di swap dalla Cache alla memoria centrale di parti di codice. Aumentando il numero degli elementi anche solo di 5 unità, le cose peggiorano drasticamente arrivando a dimensioni enormi: 240 PByte. Questo tipo di algoritmo risulterà perciò molto interessante nei casi in cui gli array da ordinare siano piccoli, nell'ordine dai 3 ai 6 elementi.

## 5.4 Innesto in algoritmi uniformi

Un'idea interessante per il nostro ambito di studio potrebbe essere quella di integrare un algoritmo non uniforme di ordinamento, ottimo su array di piccole dimensioni, all'interno di un algoritmo di ordinamento già noto e utilizzato anche per array di grandi dimensioni.

Gli algoritmi candidati, nei quali si potrebbe effettuare un innesto, sono gli algoritmi basati sul concetto di divide et impera [18]. Infatti se l'algoritmo generale ricorsivamente divide il problema dell'ordinamento in porzioni di sottoarray da ordinare, potremmo, se la lunghezza del sottoarray da ordinare è abbastanza piccola, sostituirlo con l'algoritmo di ordinamento non uniforme che termina il lavoro di ordinamento su quel piccolo sottoarray. Idealmente un algoritmo divide et impera può essere visto come un albero binario ricorsivo.

Si cerca di aumentare l'efficienza bloccando l'algoritmo divide et impera prima di arrivare alle foglie (che sarebbero un singolo elemento dell'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 sfruttano il concetto di divide et impera:

- MergeSort
- QuickSort

Il MergeSort [30] è un algoritmo con complessità teorica ottimale $\Theta(NlogN)$. Ha il 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 dopo aver ordinato nell'array temporaneo deve ricopiare le modifiche sull'array originale, il che si intuisce non esser ottimo rispetto alla somma totale delle scritture.
Il QuickSort [33], 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 $O(N^2)$, possiede un caso medio di complessità $O(Nlog_2N)$. In dettaglio i vantaggi sono i seguenti:

1. E' un algoritmo In-place, perciò non utilizza memoria secondaria durante l'esecuzione. Nella sua versione ricorsiva dovendo tener traccia delle chiamate ricorsive, utilizza una memoria pari a $log_2N$ dove N è il numero di elementi dell'array in input . 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 abbastanza 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 per essere ordinate potranno risiedere tutte completamente in memoria, senza essere costretto ad effettuare degli scambi con la memoria centrale.

4. Grazie alle sue prestazioni già collaudate nel tempo, è l'algoritmo base utilizzato dall'Introsort [26]. Questo algoritmo è un algoritmo di ordinamento ibrido ovvero che combina insieme tre algoritmi: Quicksort - HeapSort - InsertionSort. Questo algoritmo è implementato nella libreria standard dell' ANSI C, richiamabile tramite la funzione *qsort* [19], in quanto considerato un algoritmo affidabile e con buone prestazioni.

In conclusione, 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 di questo algoritmo, e l'implementazione della funzione *qsort*.

```
1 //Funzione che effettua i confronti tra gli elementi
2 int compare (const void * a, const void * b){
3    return ( *(int*)a - *(int*)b );
4 }
5
6 //Creo un vettore riempito di numeri casuali
7 void generate_vector(int v[], int nmax){
8   //Inizializzo il generatore di numeri casuali
9   srand(time(NULL));
10
11  //Inizializzo il vettore con numeri casuali
12  int i;
13  for(i=0;i<nmax;i++)
14    v[i]=rand() % nmax;
15 }
16
17 int main(int argc, char *argv[]) {
18
19  //Recupero il numero di elementi da generare
20  int nmax;
21  nmax = atoi(argv[1]);
22  int v[nmax];
23
24  //Genero il vettore da ordinare
25  generate_vector(v, nmax);
26
27  //Ordino il vettore
28  qsort( v, nmax, sizeof(int), compare );
29 }
```

Listing 5.1: Codice C dell'implementazione della funzione di ordinamento qsort

## 5.5 Implementazioni integrate

Per poter integrare all'intero del Quicksort delle funzioni non uniformi di ordinamento, abbiamo prima codificato le funzioni e poi abbiamo modificato il Quicksort affinché le potesse utilizzare. Come abbiamo già descritto nella sezione 5.3, la lunghezza dell'array non potrà essere troppo elevata, perciò si è deciso di produrre soltanto tre funzioni:

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, 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 implementazione ad albero, analoga alle due precedenti.

Facciamo notare che tutte e tre le funzioni, utilizzano gli scambi come base per l'ordinamento. Di conseguenza si è scelto di utilizzare una funzione dedicata agli scambi, per ottenere un codice il più possibile chiaro e che potesse essere facilmente comprensibile al momento di eventuali correzioni o modifiche.
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 5.2: 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 5.3: 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          }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    }else{
67      if(v[a]<v[c]){
68        if(v[c]<v[d]){
69          swap(v, a, b);
70        }else{
71          if(v[a]<v[d]){
72            swap(v, a, b);
73            swap(v, c, d);
74          }else{
75            if(v[b]<v[d]){
76              swap(v,a,b);
77              swap(v,b,d);
78              swap(v,c,d);
79
80            }else{
81              swap(v,a,c);
```

```
82                swap(v,a,d);
83              }
84            }
85          }
86        }else{
87          if(v[a]<v[d]){
88            if(v[b]<v[c]){
89              swap(v,a,b);
90              swap(v,b,c);
91            }else{
92              swap(v,a,c);
93            }
94          }else{
95            if(v[b]<v[c]){
96              if(v[c]<v[d]){
97                swap(v,a,b);
98                swap(v,b,c);
99                swap(v,c,d);
100             }else{
101               if(v[b]<v[d]){
102                 swap(v,a,b);
103                 swap(v,b,d);
104               }else{
105                 swap(v,a,d);
106               }
107             }
108           }else{
109             if(v[d]<v[c]){
110               swap(v,a,d);
111               swap(v,b,c);
112             }else{
113               if(v[b]<v[d]){
114                 swap(v,a,c);
115                 swap(v,c,d);
116               }else{
117                 swap(v,b,c);
118                 swap(v,a,b);
119                 swap(v,b,d);
120               }
121             }
122           }
123         }
124       }
125     }
126 }
```

Listing 5.4: 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 5.5: 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 5.6: 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 associare un codice binario univoco ad una permutazione tramite una serie di confronti. Una volta che è stato ottenuto il codice di una permutazione è possibile ordinarla eseguendo il minor numero di scritture, sfruttando il principio dei cicli di ordinamento. In pratica per tutte le possibili permutazioni di quella lunghezza si sono registrati in un array, 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 basterà seguire il ciclo di sostituzioni identificato dal susseguirsi di posizioni, ottenendo così l'array ordinato con il minor numero di scritture. I concetti principali sono descritti in seguito:

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 CycleSort o OlivatoCycleSort.

Questo algoritmo effettua il minor numero di scritture possibili, anche se non si riesce facilmente ad identificare la sua complessità.

#### 5.5.1.2 Descrizione dettagliata

Come abbiamo visto l'algoritmo si racchiude in due parti principali. Esistono inoltre 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 memorizzandoli in un array.

2. Applico un'euristica sui codici binari identificando la lunghezza minima necessaria affinché ogni codice risulti univoco rispetto agli altri.

3. Creo un albero binario che memorizza i codici binari, avente tutti i possibili percorsi che rappresentano la totalità dei codici binari ottimizzati. Nelle foglie di ogni percorso, e quindi per ogni codice binario, associo la chiave corretta di quella permutazione, che servirà per trovare i cicli di ordinamento precalcolati che dovranno essere eseguiti.

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

#### 5.5.1.3 Implementazione e integrazione dell'algoritmo

Seppur questo algoritmo risulti più compatto dei precedenti sono state generate implementazioni accettabili dimensionalmente solo fino ad array di lunghezza pari a 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] *
              -1)-1;
30      }
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-l == 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 5.7: Codice C del KeyHardCodedSort per 3 elementi integrato nel Quicksort

#### 5.5.1.4 Calcolo della complessità

La complessità della funzione KeyHardcodedSort può essere analizzata più precisamente suddividendola nelle sue componenti:

1. La parte di recupero del codice binario per quella combinazione, che contemporaneamente ricerca la chiave(indice nell'array) dei cicli di ordinamento da eseguire per ordinarla.

2. L'effettivo ordinamento della combinazione.

La funzione che recupera il codice binario di una combinazione è composta da una serie di confronti. Analizzando i due cicli for che effettuano i confronti si nota che partendo da sinistra verso destra, ogni elemento verrà confrontato un numero di volte pari al numero di elementi presente nel sottoarray di destra. Se consideriamo N il numero di elementi del vettore, otterremo la seguente sommatoria

$$\sum_{i=0}^{N} i = \frac{N(N-1)}{2}$$

. Dato che l'ordinamento dell'array avviene semplicemente eseguendo il ciclo di ordinamento senza effettuare ulteriori confronti, la complessità della funzione risulterà influenzata solo dalla parte di ricerca del codice binario, e sarà quindi $O(N(N-1)/2)$.

Le funzioni non uniformi che utilizzano l'albero di confronti ottimizzato5.5 e che sfruttano l'ordinamento per confronti avranno una complessità pari a $O(Nlog_2N)$ [13].

Sebbene le due complessità siano diverse, per numeri sufficientemente piccoli dell'input si possono ottenere valori della complessità comparabili.

| **Numero di elementi** | $\lceil \boldsymbol{Nlog_2(N-1)} \rceil$ | $\lceil \boldsymbol{N(N-1)/2} \rceil$ |
|---|---|---|
| 3 | 3 | 3 |
| 4 | 7 | 6 |
| 5 | 10 | 10 |
| 6 | 14 | 15 |
| 7 | 19 | 21 |

Per i valori 3, 4, 5 si nota una uguaglianza, se non addirittura un miglioramento della complessità. Questo confronto ci permette di valutare la funzione KeyHardcodedSort fino a sette elementi, mantenendo con buona approssimazione una complessità ottimale.

## 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 1. Si è cercato di produrre dei grafici il più chiari possibili, dove si evidenziano sia i miglioramenti rispetto ad una implementazione di riferimento sia i peggioramenti rispetto alla stessa.

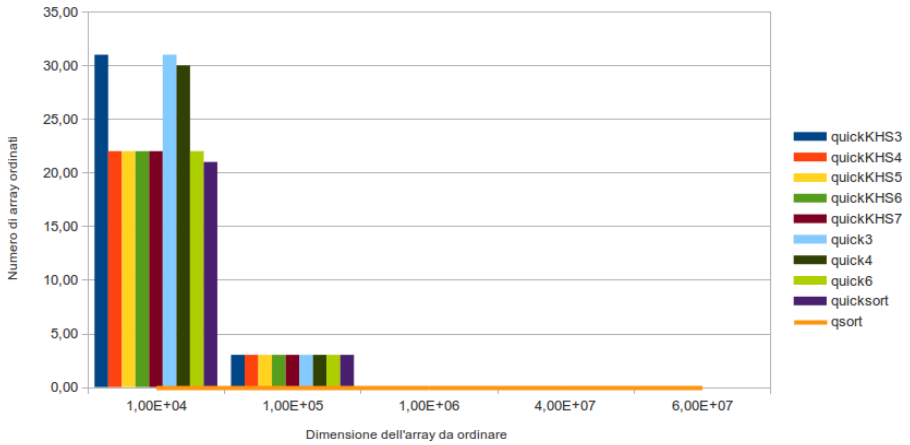


Figura 5.3: Numero totale di array ordinati prima del completo scaricamento della batteria di riferimento rispetto alla funzione qsort dell'ANSI C.

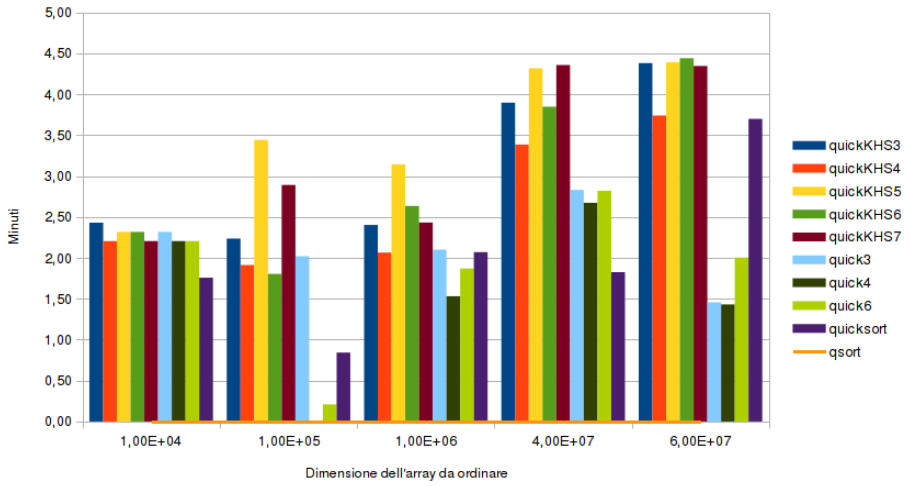


Figura 5.4: Durata della batteria di riferimento rispetto alla funzione qsort dell'ANSI C.

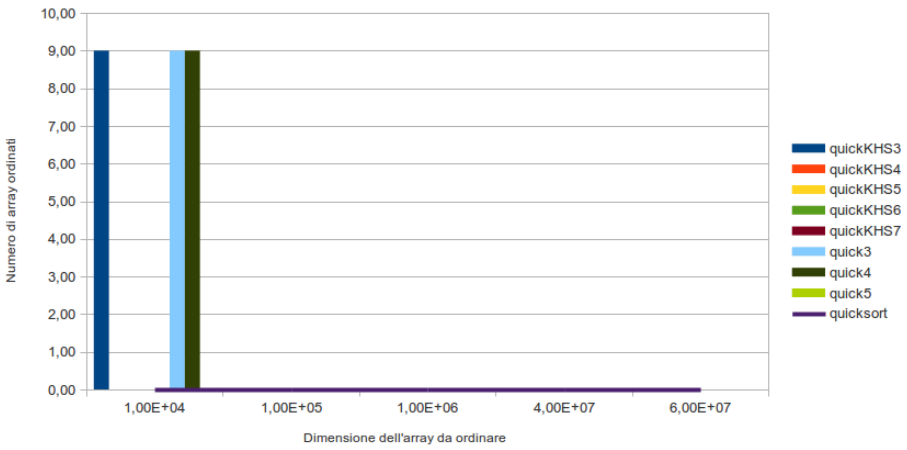


Figura 5.5: Numero totale di array ordinati prima del completo scaricamento della batteria di riferimento rispetto all'implementazione del Quicksort.

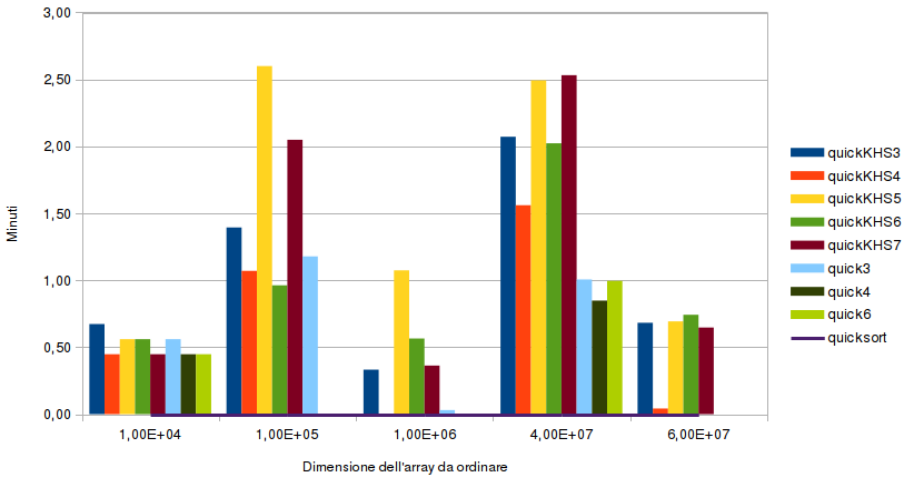


Figura 5.6: Differenze nella durata della batteria di riferimento rispetto all'implementazione del Quicksort.

É possibile notare che per quanto riguarda la vita della batteria, l'implementazione del Quicksort con KeyHardcodedSort a 5 elementi (quickKHS5) risulterebbe essere la migliore, sia nei confronti dell'implementazione *qsort*, sia del Quicksort nelle sue varianti.

Dal punto di vista dell'efficienza computazionale notiamo che gli algoritmi proposti, anche se molti sembrerebbero migliorare la durata di vita della batteria, pochi riescono ad essere anche più efficienti, ovvero riescono ad

ordinare un numero maggiore di array rispetto agli algoritmi standard. Gli unici a riuscire nell'intento risulterebbero essere l'implementazione Quicksort + Threesort (quick3), Quicksort + KeyHardcodedSort (quickKHS3) e Quicksort + Foursort (quick4). Queste versioni mostrerebbero di essere in grado di ordinare un numero maggiore di array rispetto agli algoritmi standard per array di lunghezza piccola, ed anche in grado di consumare meno energia, allungando la durata di vita della batteria di riferimento.

# Capitolo 6

# Conclusioni

## 6.1 Conclusione e prospettive future

Abbiamo visto nei capitoli precedenti come lo studio del consumo energetico in ambito software possa fornire dei risultati di interesse e potrebbe proporre miglioramenti algoritmici altrimenti poco considerati.

Il primo risultato ottenuto risulterebbe la riconferma dello Standard swap come l'algoritmo di scambio più energeticamente efficiente e consigliato in generale per ogni tipo di utilizzo.

Il secondo risultato cerca di analizzare gli algoritmi con il minor numero di scritture, con lo studio di un algoritmo ideato in proposito, il CycleSort e di un algoritmo che cerca di proporre un miglioramento, l'OlivatoCycleSort. L'approfondimento su questo caso di studio aprirebbe orizzonti ad aspetti sconosciuti, decisamente più proiettati in una visione futura sia per quanto riguarda la problematica algoritmica sia quella architetturale.

Il terzo e più interessante risultato, presenta i miglioramenti effettuati nell'algoritmo di ordinamento di riferimento per il linguaggio ANSI C. Una modifica algoritmica che tenga conto dei parametri di incidenza sul consumo sembrerebbe ottenere risultati misurabili,sia in una differenza nella durata di vita della batteria sia per quanto riguarda l'efficienza.

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 non 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.

Le versioni testate sono state molte, sia per dare un ampio spettro dei risultati che si potrebbero ottenere attraverso le modifiche, sia per trovare un algoritmo modificato che si possa presentare ad essere più efficiente su ogni aspetto ricercato. Quest'ultimo potrebbe diventare di interesse implementativo e strutturale in quegli algoritmi che si basano sull'ordinamento, o sulla ripetizione ciclica di ordinamenti, aiutando ad ottenere dei risultati utili sul consumo energetico totale.

Ricordiamo 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. Consapevolmente si sono limitati i contenuti alle informazioni più essenziali. Le molte ulteriori considerazioni potrebbero essere riprese più avanti, in quanto l'argomento, se trattato con la assoluta certezza, avrebbe comportato una spesa di tempo e risorse molto al di sopra delle normali possibilità. Nonostante ciò si spera che i risultati ottenuti possano essere rilevanti e utili per incoraggiare un'ulteriore investimento nella ricerca algoritmica come chiave dello sviluppo energetico futuro.

# Bibliografia


[1] *Approssimazione di Stirling*. 2016. URL: https://en.wikipedia.org/wiki/Stirling%27s_approximation.

[2] *ARM Architecture*. 2016. URL: https://en.wikipedia.org/wiki/ARM_architecture.

[3] *ARM Coretex-A7*. 2015. URL: https://en.wikipedia.org/wiki/ARM_Cortex-A7.

[4] *Atmel AVR Microontroller*. 2016. URL: https://en.wikipedia.org/wiki/Atmel_AVR#Device_overview.

[5] *Atmel AVR STK500*. 2016. URL: http://www.atmel.com/tools/STK500.aspx.

[6] *Bitwise XOR*. 2016. URL: https://en.wikipedia.org/wiki/Bitwise_operation#XOR.

[7] *Broadcom BCM2836*. 2016. URL: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/.

[8] Christian Bunse et al. «Choosing the" Best" Sorting Algorithm for Optimal Energy Consumption.» In: *ICSOFT (2)*. 2009, pp. 199–206.

[9] Christian Bunse et al. «Exploring the Energy Consumption of Data Sorting Algorithms in Embedded and Mobile Environments». In: *2014 IEEE 15th International Conference on Mobile Data Management* 0 (2009), pp. 600–607. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.543.8109&rep=rep1&type=pdf.

[10] *Cache levels properties*. URL: http://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips.

[11] *Cache miss*. 2015. URL: https://en.wikipedia.org/wiki/CPU_cache#Cache_miss.

[12] *Calcolatore quantistico*. 2015. URL: https://en.wikipedia.org/wiki/Quantum_computing.

[13] *Comparison sort computational complexity*. 2015. URL: https://en.wikipedia.org/wiki/Comparison_sort#Performance_limits_and_advantages_of_different_sorting_techniques.

[14] *Counting Sort*. 2015. URL: https://en.wikipedia.org/wiki/Counting_sort.

[15] *Cyclesort*. 2015. URL: https://en.wikipedia.org/wiki/Cycle_sort.

[16] *Cyclic Permutation*. 2015. URL: https://en.wikipedia.org/wiki/Cyclic_permutation.

[17] *Debian Limits*. Debian Project. 2015. URL: https://wiki.debian.org/it/Limits.

[18] *Divide et Impera*. 2015. URL: https://en.wikipedia.org/wiki/Divide_and_conquer_algorithms.

[19] *Funzione qsort ANSI C*. 2015. URL: https://en.wikipedia.org/wiki/Qsort.

[20] *GCC 4.8 Release Series*. Free Software Foundation, Inc. 2015. URL: https://gcc.gnu.org/gcc-4.8/.

[21] *GCC 5 Release Series Changes, New Features, and Fixes*. Free Software Foundation, Inc. 2016. URL: https://gcc.gnu.org/gcc-5/changes.html.

[22] *Gestione della memoria*. 2015. URL: http://vision.unipv.it/corsi/SistemiOperativi/lucidi/SO-12.pdf.

[23] Mark Gottscho, Dr. Puneet Gupta e Abde Ali Kagalwalla. *Analyzing Power Variability of DDR3 Dual Inline Memory Modules*. NanoCAD Lab, UCLA Electrical Engineering. URL: http://nanocad.ee.ucla.edu/pub/Main/Publications/UG2_paper.pdf.

[24] 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.

[25] *Insertionsort*. 2015. URL: https://en.wikipedia.org/wiki/Insertion_sort.

[26] *Introsort*. 2015. URL: https://en.wikipedia.org/wiki/Quicksort.

[27] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0-201-89685-0.

[28] *Landauer's principle*. 2015. URL: https://en.wikipedia.org/wiki/Landauer%27s_principle.

[29] *Memory Management*. 2015. URL: https://en.wikipedia.org/wiki/Memory_management_(operating_systems).

[30] *Mergesort*. 2015. URL: https://en.wikipedia.org/wiki/Merge_sort.

[31] *Mobile Marketing Statistics compilation*. 2015. URL: http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/.

[32] *Program Counter*. 2016. URL: https://en.wikipedia.org/wiki/Program_counter.

[33] *Quicksort*. 2015. URL: https://en.wikipedia.org/wiki/Introsort.

[34] *Radioisotope Power System*. National Aeronautics e Space Administration. 2015. URL: http://solarsystem.nasa.gov/rps/home.cfm.

[35] *Radixsort*. 2016. URL: https://en.wikipedia.org/wiki/Radix_sort.

[36] *Raspberry Pi 2 Model B*. 2015. URL: https://www.raspberrypi.org/products/raspberry-pi-2-model-b/.

[37] *Raspbian Jessie*. 2015. URL: https://www.raspberrypi.org/blog/raspbian-jessie-is-here/.

[38] *Registri del processore*. 2015. URL: https://en.wikipedia.org/wiki/Processor_register.

[39] Euiseong Seo, Seon Yeong Park e Bhuvan Urgaonkar. «Empirical Analysis on Energy Efficiency of Flash-based SSDs». In: *Proceedings of the 2008 Conference on Power Aware Computing and Systems*. HotPower'08. San Diego, California: USENIX Association, 2008, pp. 17–17. URL: http://dl.acm.org/citation.cfm?id=1855610.1855627.

[40] *Standard Swap*. 2015. URL: https://en.wikipedia.org/wiki/Standard_swap.

[41] *Swap throught addition and subtraction*. 2015. URL: https://en.wikipedia.org/wiki/Swap_(computer_science)#Swap_through_addition_and_subtraction.

[42] *Time command Unix*. The IEEE e The Open Group. 2015. URL: http://pubs.opengroup.org/onlinepubs/000095399/utilities/time.html.

[43] Guy Webster. *Computer Swap on Curiosity Rover*. National Aeronautics e Space Administration. 2013. URL: http://www.nasa.gov/mission_pages/msl/news/msl20130228.html.

[44] *Xor Swap*. 2015. URL: https://en.wikipedia.org/wiki/XOR_swap_algorithm.