SAPIENZA
UNIVERSITÀ DI ROMA

Programmazione funzionale: analisi computazionali di programmi che generano strutture dati infinite

Facoltà di Ingegneria dell'informazione, Informatica e Statistica

Corso di Laurea in Informatica

Candidato

Gianmarco Picarella
Matricola 1788997

Relatore

Prof. Ivano Salvo

Anno Accademico 2019/2020

Sommario

Disporre di algoritmi efficienti per il calcolo di sequenze di numeri primi è oramai un requisito chiave in molteplici realtà industriali ed accademiche. Il paradigma di programmazione funzionale, offre un approccio alla risoluzione di problemi computazionali attraverso una modellazione matematica basata su funzioni. Lo scopo di questa tesi, è quello di investigare da un punto di vista teorico ed implementativo, il problema del calcolo di sequenze di numeri primi in Haskell. Per calcolo di numeri primi, intendiamo l'estrapolazione, a partire da un intervallo $[2..n]$, della sequenza crescente di numeri primi $[2, 3, .., p_{\pi(n)}]$. Partendo con una breve introduzione ai linguaggi funzionali, e, ponendo particolare enfasi su Haskell, presentiamo al lettore, attraverso il caso di studio dell'ordinamento, i principali strumenti di benchmarking e profiling utilizzati. Successivamente, dopo aver presentato formalmente gli algoritmi funzionali scelti e sfruttando alcuni dei risultati ottenuti dal teorema dei numeri primi, forniremo dei limiti superiori al numero di operazioni eseguite dai suddetti. Infine, attraverso una serie di test pratici, eleggeremo l'algoritmo funzionale per il calcolo di sequenze di numeri primi più efficiente e resistente, tra quelli proposti, al degrado prestazionale.

Indice

Capitolo 1

Introduzione

Sebbene LISP, il primo linguaggio funzionale, risalga al 1958, il paradigma imperativo ha dominato, per decenni, il panorama industriale ed accademico nel mondo dell'informatica. Proprio per questo, nella prima parte di questo capitolo, inizieremo con una breve introduzione sull'universo funzionale, ponendo particolare enfasi sulle proprietà che Haskell mette a disposizione. Avremo modo di introdurre il lettore ad una filosofia di programmazione con cui modellare ed affrontare matematicamente problemi computazionali come quelli introdotti nel secondo capitolo. Successivamente, attraverso il caso di studio dell'ordinamento, studieremo le differenze prestazionali tra implementazione imperativa e funzionale, introducendo, al contempo, il lettore all'insieme di strumenti di benchmarking e profiling utilizzati nel corso di questo studio, utili ad analizzare più in profondità programmi funzionali il cui comportamento, soprattutto per programmatori naive, può risultare spesse volte di difficile comprensione.

1.1 Un entry point funzionale

Un programma imperativo è un insieme di istruzioni o statement eseguiti sequenzialmente, i quali, modificano la memoria del calcolatore dedicata alla loro esecuzione: contare quanti quadrati perfetti sono presenti all'interno di una lista, è possibile attraverso l'utilizzo di una variabile che funge da contatore, o più astrattamente, stato. Questo modello di computazione ha dominato il mondo dell'informatica fino ad oggi poiché rispecchia esattamente il modello operativo hardware. Sostanzialmente è grazie a questa mappatura tra architettura sottostante della macchina e software, che senza particolari difficoltà, siamo in grado di calcolare l'andamento asintotico dell'esecuzione software. A seguito dei progressi ottenuti nella teoria del lambda calcolo, ovvero, un sistema matematico formale che esprime una computazione attraverso l'applicazione, concatenazione e sostituzione di funzioni, è stato definito un paradigma di programmazione. Con concatenazione di funzioni, intendiamo l'applicazione a cascata di più funzioni sequenzialmente, ovvero, l'applicazione della funzione successiva al risultato prodotto dalla precedente. La programmazione funzionale è quindi la manifestazione pratica del lambda calcolo tipato, ovvero, un paradigma di programmazione in cui un problema computazionale è modellato attraverso un'espressione da valutare, composta da funzioni pure e loro concatenazioni, in cui l'accento

viene posto maggiormente nulla definizione delle funzioni e non sulla sequenza di valori che devono calcolare. Proponiamo ora un programma imperativo in C che data una lista di interi, restituisce il numero di quadrati perfetti presenti. Subito dopo, seguendo una logica funzionale, riportiamo la sua traduzione in Haskell, il linguaggio funzionale di riferimento in questa tesi. Come si può notare, la sintassi funzionale è molto più compatta ed il suo potere espressivo consente di concentrare il core del nostro programma in poco spazio.

```c
int countPerfectSquares(int * l, int size) {
    int counter = 0;
    for(int i = 0; i < size; i++) {
        int step = 1;
        while (step*step < l[i] && (step++));
        if(step*step == l[i]) counter++;
    }
    return counter;
}
```

Listing 1.1: Implementazione C per il conteggio dei quadrati perfetti in un array

Come accennato in precedenza, la programmazione funzionale si riduce a fornire una descrizione della funzione e quindi dell'oggetto che si vuole computare. In questo esempio, la funzione *cqp* invoca la funzione *cqp′* con un secondo parametro, che funge da accumulatore invisibile allo scope esterno. Contare il numero dei quadrati perfetti in una lista, equivale a verificare ricorsivamente se l'elemento x_i ha un quadrato perfetto con $0 \leq i < len(xs)$ incrementando il contatore, se opportuno. Dopo questa breve introduzione, passiamo ad un'analisi più approfondita delle proprietà fornite da un linguaggio funzionale.

```haskell
cqp xs = cqp' xs 0 where
  cqp' [] a = a
  cqp' (x:xs) a
    | x == head (dropWhile (<x) sqr x) = cqp' xs (a+1)
    | otherwise = cqp' xs a
  sqr x = map (^2) [1..x]
```

Listing 1.2: Implementazione Haskell per il conteggio dei quadrati perfetti in una lista

1.1.1 Il building block funzionale: le funzioni pure

Le vere protagoniste che fungono da elemento costituente del paradigma funzionale, sono le funzioni pure. Una funzione, è definita pura se il suo valore di ritorno, dipende esclusivamente dai parametri inferiti dall'esterno, senza side effects osservabili. La conseguenza immediata è che i programmi realizzati tramite linguaggi funzionali, sono altamente parallelizzabili. L'elemento principale in un programma funzionale è la funzione. Applicando più funzioni a cascata, siamo in grado di descrivere comportamenti complessi attraverso l'uso di porzioni di codice elementare. A differenza dei linguaggi imperativi, in cui ogni funzione corrisponde ad un insieme di statement, la definizione di una funzione, è rappresentata da un albero di espressioni, ognuno

dei quali produce un valore finale attraverso un insieme di passi chiamati riduzioni. Un'espressione *expr* è definita redex se è possibile effettuare una o più riduzioni su una delle sue sottoespressioni. D'altra parte, diremo che *expr* è espansa o in forma normale se non è possibile procedere con ulteriori passi di riduzione (ed è quindi un valore). Riportiamo per chiarezza un toy example riguardo una semplice riduzione.

$$\begin{aligned} expr &= sum(12, div(exp(10), 2)) \\ &= 12 + div(exp(10), 2) \\ &= 12 + exp(10)/2 \\ &= 12 + e^{10}/2 \\ &= 12 + 21364 \\ &= 21376 \end{aligned} \tag{1.1}$$

Nella riduzione che abbiamo riportato, sono state introdotte delle funzioni matematiche largamente conosciute, in modo da poter evitare la loro definizione. Ad ogni passo, riduciamo una sottoespressione procedendo per ordine esterno (vedremo a seguito il perché) finché *expr* non sarà in forma normale.

1.1.2 Riduzione graph outermost

È necessario porre particolare attenzione alla strategia di riduzione: esistono molteplici espressioni, la cui terminazione e numero di passi necessari alla riduzione, dipendono dall'ordine di valutazione dei termini che la costituiscono. Consideriamo ora le seguenti definizioni di funzioni e l'espressione *expr*.

$$\begin{aligned} &loop\ 0 = 0 + loop\ 0 \\ &loop\ 1 = 1 + loop\ 1 \\ &first\ a\ b = a \\ &expr = first(square(3 \cdot 2),\ loop\ 1) \end{aligned} \tag{1.2}$$

Non a caso, abbiamo introdotto due definizioni con parametri differenti per la funzione *loop*. Quello che accade è interessante: Haskell mantiene entrambe le definizioni e applicando una tecnica di pattern matching, seguendo l'ordine di definizione, deciderà quale delle due espressioni usare. Ad ogni passo di riduzione, quindi, la sostituzione di una funzione con la sua espressione, viene eseguita a seguito di un match nella sintassi. Riducendo *expr* secondo una logica strict, *expr* risulterebbe $\perp$ o undefined: il secondo membro *loop*1 della tupla, definisce se stesso ricorsivamente, non raggiungendo quindi una riduzione in forma normale in un numero finito di passi. Ciononostante, evitando la riduzione del secondo parametro, saremmo comunque in grado di ridurre l'espressione *expr* in forma normale: dopo la riduzione di $first$, l'unico elemento sopravvissuto è infatti il primo elemento della tupla.

$$\begin{aligned} expr &= first(square(3 \cdot 2),\ loop\ 1) \\ &= square(3 \cdot 2) \\ &= (3 \cdot 2) \cdot (3 \cdot 2) \\ &= 6 \cdot (3 \cdot 2) \\ &= 6 \cdot 6 \\ &= 36 \end{aligned} \tag{1.3}$$

La strategia di riduzione outermost, esegue le riduzioni a partire dalla funzione più esterna, valutando solo i parametri necessari. Inoltre, una riduzione secondo la strategia outermost, soddisfa la seguente proprietà: se esiste un ordine di passi con cui la riduzione termina, allora anche la riduzione outermost terminerà. La strategia di riduzione outermost da sola non è sufficiente per garantire una procedura di riduzione efficiente: valutando solo in un secondo momento i parametri di una funzione, è possibile che la stessa sequenza di riduzioni venga eseguita più volte come in $expr = (3 \cdot 2) \cdot (3 \cdot 2)$ e con un incremento esponenziale in memoria e passi di riduzione nel caso peggiore. Per questo motivo, ogni espressione ripetuta, viene ridotta una sola volta e sostituita in tutte le sue occorrenze future. La tecnica è chiamata outermost graph reduction ed è alla base del sistema di riduzione del compilatore Haskell GHC.

1.1.3 Valutazione lazy

Un concetto a cui siamo molto legati nel mondo imperativo è la valutazione eager: un'espressione, viene valutata non appena è assegnata ad una variabile. Valutare con una tecnica eager, permette di prevedere la quantità di lavoro che verrà svolto da una porzione di software, in modo veloce e preciso. Introduciamo ora una seconda strategia di valutazione denominata Lazy Evaluation, letteralmente "valutazione pigra". Il concetto alla base è quello di eseguire un passo di calcolo, solo se strettamente necessario al raggiungimento dell'output finale. Qui sorge un secondo interessante spunto: perché mai calcolare un dato, se questo non viene comunicato al mondo esterno? Perché mai valutare il parametro di una funzione, se questo non sarà mai utilizzato? In effetti, abbiamo già visto che, tramite la riduzione outermost, valutiamo solo i parametri necessari, risparmiando quindi lavoro inutile al raggiungimento del risultato. La valutazione lazy, introduce un sistema di packaging il cui elemento costituente è il Thunk. Un Thunk racchiude un'espressione non ancora valutata, che probabilmente, verrà eseguita in futuro. Un'espressione, a sua volta, può contenere dei Thunk a sostituzione di sottoespressioni non ancora espanse. Un Thunk non effettua l'intera riduzione, bensì esegue solo il lavoro necessario ai fini della valutazione dell'espressione padre. Mostriamo a seguito un esempio pratico in cui valutiamo l'espressione $expr = (square(3 \cdot 2), 1)$.

$$\begin{aligned} expr &= THUNK \\ &= (THUNK, THUNK) \\ &= (square(THUNK), THUNK) \\ &= (square(6), THUNK) \\ &= (6 \cdot 6, THUNK) \\ &= (36, 1) \end{aligned} \tag{1.4}$$

Al secondo passo di riduzione, siamo di fronte ad un'espressione in Weak head normal form, ovvero, un'espressione il cui costruttore più esterno è stato valutato (Tuple constructor in questo caso) ma le sottoespressioni potrebbero non esserlo. Di conseguenza, un'espressione in forma normale è anche in Weak head normal form, ma non viceversa. In situazioni più complesse, l'ordine di valutazione lazy è spesso difficile da tracciare e prevedere per un umano, e per questo, ancora oggi non esiste una strategia semplice e ben definita con cui provare per via teorica le performance

computazionali, e di memoria, di una procedura lazy. Una delle conseguenze della valutazione lazy, è la possibilità di definire e maneggiare strutture dati infinite: valutando un elemento alla volta, scartandolo quando non è più necessario, è possibile usufruire di liste infinite con un minimo utilizzo di memoria. Nel prossimo capitolo vedremo come sfruttare le liste infinite nella generazione di sequenze di numeri primi.

1.1.4 Funzioni di ordine superiore

Una necessità che sorge molto frequentemente nella programmazione funzionale, è quella di applicare delle trasformazioni ad oggetti complessi, come per esempio le liste. Sebbene la procedura ricorsiva, con cui scorrere la struttura dati, sia sempre la stessa, la trasformazione applicata ad ogni suo elemento cambia in funzione del risultato che vogliamo ottenere. Le funzioni Higher Order, permettono di inferire attraverso dei parametri, altre funzioni, modificando quindi il comportamento della procedura. Esse compongono uno degli elementi chiave della programmazione funzionale. Un esempio di funzione HO è $map\ f\ xs$ la quale, data una lista xs, applica la funzione f ad ogni suo elemento, eseguendo quindi una trasformazione sulla lista. Poc'anzi abbiamo accennato che, differentemente dai linguaggi imperativi, una funzione è stateless e non presenta side effects. Dipendendo solo ed esclusivamente dai parametri immessi, siamo certi che nessuna variabile esterna allo scope, possa influenzare la computazione di una procedura funzionale. La ricorsione è un altro argomento cardine nel mondo funzionale: a differenza dei linguaggi imperativi, in cui possiamo alterare lo stato del programma, nella programmazione funzionale questo è negato. L'implicazione immediata è l'assenza di for e $while$ e l'utilizzo della ricorsione come unico strumento di controllo della computazione. La funzione cqp' è un esempio di come sfruttare la ricorsione per attraversare una lista, ed incarna esattamente la metodologia descritta poc'anzi.

1.1.5 Haskell come arrivo

Il linguaggio che implementa tutte le caratteristiche fino ad ora discusse è Haskell, uno dei linguaggi più affermati nell'universo funzionale. Alcune caratteristiche del linguaggio, sono, oltre a quelle precedentemente descritte, la type inference e safety: le incongruenze di tipi che porterebbero ad errori a runtime vengono scovate a tempo di compilazione e la possibilità di null pointer viene praticamente eliminata. È importante notare che Haskell, nonostante sia un linguaggio ad alto livello, è estremamente efficiente: il compilatore GHC è in grado, in molteplici situazioni, di generare un eseguibile con prestazioni vicine ai suoi corrispettivi imperativi. Il vantaggio risiede proprio nel codice ad alto livello: il compilatore è in grado di eseguire ottimizzazioni estreme, modificando radicalmente la struttura del codice compilato. Per un approfondimento del linguaggio ed il suo ecosistema, rimandiamo alla Haskell Wiki [Pag20a].

1.2 Il problema dell'ordinamento

Il problema dell'ordinamento è un ottimo caso di studio per mostrare le differenze tra i due paradigmi di programmazione. In questa parte, mostreremo come un

algoritmo, a parità di complessità asintotica, possa stravolgere le sue performance se modellato con paradigmi differenti. Il caso che analizzeremo è quello dell'algoritmo Quicksort, confrontando le implementazioni C ed Haskell attraverso l'uso di strumenti di benchmarking che introdurremo a breve. La scelta dell'algoritmo non è casuale ed apparirà più chiara nelle sezioni a seguire.

1.2.1 Quicksort

Data una lista di interi da ordinare xs, l'algoritmo di Quicksort sceglie ad ogni passo k-esimo un elemento p_k (pivot) presente in xs attraverso una strategia ben definita (solitamente attraverso una procedura casuale). Successivamente, suddivide xs in due sottoliste $xs_{\leq p_k}, xs_{>p_k}$ (nella prima lista non consideriamo il pivot p_k scelto) contenenti rispettivamente tutti gli elementi minori o uguali e maggiori a p_k. Infine, assegna il pivot p_k alla sua posizione definitiva all'indice $len(xs_{\leq p_k})$, per poi richiamare ricorsivamente se stessa sulle due nuove sottoliste generate e ripetere la procedura fintanto che $len(xs_{<}p_k) > 1 \vee len(xs_{>}p_k) > 1$. Ad ogni passo, il pivot può considerarsi ordinato. Il metodo con cui scegliere il pivot risulta cruciale per ottenere una complessità asintotica ottimale nel caso medio. Per esempio, scegliendo sempre il primo elemento come pivot, nel caso di una lista decrescente, la complessità esplode fino a raggiungere $O(n^2)$. Formalmente possiamo derivare il caso peggiore seguendo l'equazione di ricorrenza

$$\begin{aligned} T(n) &= T(n-1) + O(n) \\ T(1) &= O(1) \end{aligned} \tag{1.5}$$

Tramite srotolamento otteniamo $T(n) = O(n)+O(n-1)+\ldots+O(1)$ ovvero $O(n^2)$. I punti di forza dell'algoritmo sono molteplici: la possibilità di scegliere casualmente il pivot, permette nel caso medio, di contenere il numero di passi nell'ordine di Mergesort mentre l'assegnamento sul posto consente di non allocare memoria aggiuntiva.

```
void quickSort(int * nums, int s, int e){
    if (s >= e) return;
    int pivot = nums[(s + e) / 2] i = s-1, j = e+1;
    while (true){
        while (nums[++i] < pivot);
        while (nums[--j] > pivot);
        if (i >= j) break;
        int t = nums[j];
        nums[j] = nums[i];
        nums[i] = t;
    }
    quickSort(nums, s, j);
    quickSort(nums, j + 1, e);
}
```

Listing 1.3: Implementazione C dell'algoritmo di Quicksort

Scegliendo casualmente il pivot n volte, siamo certi che circa $\frac{n}{2}$ volte, p_k è compreso nel 50% della lista centrata all'indice $\frac{len(xs)}{2}$. Questo permette, con un numero di passi limitato a $O(log(\frac{4n}{3}))$, di ridurre la lista ad un solo elemento. Considerando

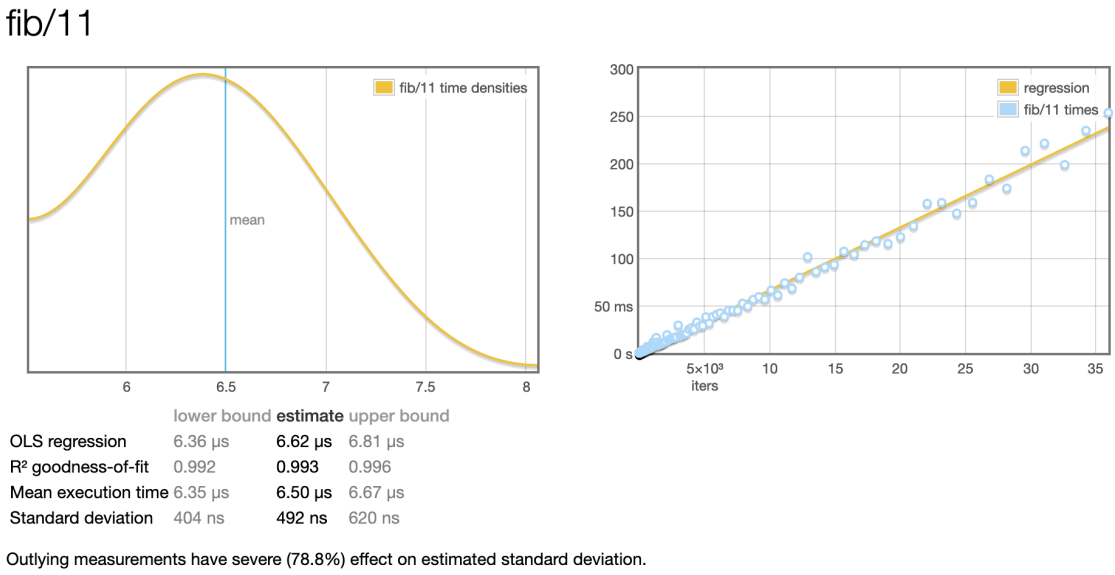
che ad ogni passo k-esimo vengono eseguite $O(n)$ operazioni, otteniamo che il limite superiore al caso medio è uguale a $O(nlog(n))$. Faremo riferimento a questa versione imperativa di Quicksort per il resto del capitolo.

1.3 I tools utilizzati

Nel corso di questo studio, utilizzeremo alcuni dei più importanti strumenti di benchmarking a disposizione per Haskell. Siamo interessati alla generazione di input per i nostri algoritmi, l'esecuzione di benchmark su di essi, e la comparazione sistematica dei dati ottenuti. Parallelamente, tramite il profiler integrato in GHC, durante l'esecuzione, monitoreremo la memoria utilizzata.

1.3.1 Criterion

Criterion è una libreria Haskell che fornisce un sistema di benchmarking basato su unità di esecuzione isolate da cui ottenere, a fine computazione, dei report in formato Html visualizzabili tramite web browser. I dati restituiti in un report di Criterion sono molteplici: a seguire, procederemo ad una loro introduzione.



	lower bound	estimate	upper bound
OLS regression	6.36 µs	6.62 µs	6.81 µs
R^2 goodness-of-fit	0.992	0.993	0.996
Mean execution time	6.35 µs	6.50 µs	6.67 µs
Standard deviation	404 ns	492 ns	620 ns

Outlying measurements have severe (78.8%) effect on estimated standard deviation.

Figura 1.1: Esempio di report Html generato da Criterion

Misuramenti aggregati

Durante la fase di benchmarking, Criterion esegue, in un arco prestabilito di tempo, più volte la stessa istanza di test con gli stessi input (persino migliaia a patto che il tempo di esecuzione unitario lo permetta) restituendo il tempo medio di esecuzione, la sua deviazione standard ed una approssimazione della funzione KDE (grafico a destra). Il motivo per cui eseguire più volte lo stesso test, deriva dalla necessità di ridurre il disturbo sulle misurazioni eseguite. Le cause di disturbo esterno possono essere molteplici: le più comuni sono ad esempio, il cpu scaling o l'esecuzione in

background di processi. Per determinare quanto i fattori di disturbo abbiano inficiato sulle misurazioni, Criterion fornisce il valore "variance introduced by outliers" il quale indica in percentuale la quantità di disturbo introdotto. Solitamente, un valore inferiore al 10% è considerato accettabile.

La funzione KDE

Attraverso i tempi di esecuzione raccolti, Criterion calcola la funzione di densità di probabilità o KDE (grafico a sinistra). Esso indica, per una qualsiasi esecuzione, la probabilità di ottenere un determinato tempo di esecuzione espresso in nanosecondi. Il KDE viene rappresentato come una curva su cui viene inoltre specificato il valore medio. Spesso, viene utilizzato un grafico ad istogramma per delineare la curva della funzione di densità di probabilità. Per ottenere una buona approssimazione, e di conseguenza un grafico attendibile, è necessario scegliere il giusto numero di cestini in cui mappare i tempi ottenuti. Questa procedura, se non automatizzata, risulta essere poco scalabile ed altamente time consuming.

Regressione OLS

Nel calcolo dei grafici sopra citati sfruttiamo la linearità della variabile indipendente I, ovvero, le iterazioni eseguite sulla procedura in analisi, rispetto alla variabile dipendente T, ovvero, il tempo di esecuzione espresso in nanosecondi. Di conseguenza, per calcolare una funzione di densità probabilistica, è necessario stimare la dipendenza funzionale tra le due variabili aleatorie. Criterion stima la dipendenza tra I e T tramite la tecnica di regressione lineare: essa consiste nel trovare un insieme di beta ottimali con cui calcolare esplicitamente l'equazione della retta che approssima la dipendenza funzionale tra le due variabili I e T. In generale, ponendo $\epsilon = \sum_{i=0}^{n}(\hat{y} - y_i)^2$ come la somma dell'errore quadratico ad ogni campione, il problema di regressione lineare OLS, per 2 variabili casuali, si riduce a trovare la coppia $(\beta_0,\ \beta_1)$ tale che la retta di equazione $\hat{y}_i = \beta_0 + \beta_1 x_i$ abbia ϵ minimale, ovvero che la retta approssimi la dipendenza esistente con il minore errore possibile. Il valore di "OLS Regression" fornito da Criterion, è dunque calcolato tramite una media, seguendo i valori forniti da $\hat{y}$ in modo tale da ottenere un valore preciso che non tenga conto di fattori di disturbo costanti come ad esempio l'overhead prodotto dai test. Tramite il rapporto $g = \frac{\sum_{i=0}^{n}(y_i - \bar{y})^2}{\sum_{i=0}^{n}(\hat{y}_i - \bar{y})^2}$ cioè il rapporto tra la distanza dal valore medio $\bar{y}$ dei campioni y_i e $\hat{y}_i$, Criterion fornisce il valore di "Goodness of fit", ovvero la precisione con cui y è approssimata da $\hat{y}$.

1.3.2 Quickcheck

Quickcheck è un tool di validazione software dedicato alla generazione di test cases. Nonostante QuickCheck offra un meccanismo automatico per generare input, tramite type inference, è possibile generare istanze di input attraverso la propria implementazione.

```
class Arbitrary a where
  arbitrary :: Gen a
```

Listing 1.4: Classe Arbitrary definita in Quickcheck

La classe fornita da QuickCheck permette di creare un'istanza *Arbitrary a*, con *a* equivalente al tipo in input alla funzione da testare. L'istanza avrà associata una funzione *arbitrary* :: *Gen a*, che restituisce un *Gen a*, ovvero, un generatore di elementi di tipo *a*. Da notare inoltre, come molto spesso risulti necessario eseguire dei test su istanze di dimensione crescente o comunque variabile. A tale scopo la primitiva QuickCheck $sized :: (Int -> Gen\ a) -> Gen\ a$ consente di generare un input con dimensione correlata alla variabile parametrica n.

1.3.3 Profiler GHC

Parallelamente ai tempi di elaborazione, come già accennato, siamo interessati alla massima quantità di memoria allocata durante l'esecuzione. Proprio per questo, il terzo strumento che utilizzeremo, è il profiler fornito da GHC. Il profiler GHC utilizza un sistema basato su cost centres, ovvero, una serie di annotazioni necessarie al tracciamento della memoria allocata. Ad ogni espressione, è quindi associato un cost centre insieme al relativo costo espresso in termini di memoria allocata. La struttura dei cost centres è quindi riconducibile ad uno stack in cui il costo di ogni cost centre è la somma dei costi dei figli.

```
{-# SCC "name" #-} <expression>
```

Listing 1.5: Sintassi Haskell di un cost centre

Tuttavia è necessario, in fase di compilazione, specificare al compilatore la modalità profiling e l'utilizzo delle annotazioni tramite i flag -prof (profiling) e -fprof-auto (aggiunta annotazioni automatiche). I dati prodotti dal profiling vengono salvati, a fine computazione, all'interno di un file di report *prog.hp* che è possibile convertire in un grafico tramite l'utility presente in Ubuntu, *hp2ps*. Ogni file di report .hp, tramite l'utilizzo dei cost centres, riporta le risorse dedicate ad ogni porzione di codice nell'arco dell'intera esecuzione.

1.3.4 AutoBench

Quello a cui siamo realmente interessati, è un software in grado di automatizzare l'intero processo svolto dagli strumenti sopracitati fornendo al contempo le stesse informazioni reperibili da un report generato da Criterion. AutoBench si occupa proprio di questo: utilizza Quickcheck per generare le istanze di input, validare le procedure da testare, e Criterion per l'analisi a tempo di esecuzione.

```
def :: TestSuite
def =
   { _progs     = []
   , _dataOpts  = def
   , _analOpts  = def
   , _critCfg   = Criterion.Main.Options.defaultConfig
   , _baseline  = False
   , _nf        = True
   , _ghcFlags  = []
   }
```

Listing 1.6: Criterion setup object

AutoBench, permette la creazione di TestSuites, ovvero, unità isolate in cui eseguire test multipli su più procedure. Essi risultano particolarmente utili quando, ad esempio, sorge la necessità di raggruppare più procedure per grado di ottimizzazione o per task eseguito. Ogni TestSuite è definito da una struttura con più parametri personalizzabili in base alle proprie esigenze.

- _dataOpts permette all'utente di fornire i propri input da testare (tramite il comando *Manual "..."*) oppure generarli casualmente (tramite *Gen*).
- _progs mantiene la lista delle funzioni da testare.
- _analOpts mantiene le impostazioni per l'analisi statistica dei dati.
- _critCfg mantiene le impostazioni di Criterion.
- _nf indica se i test cases devono essere ridotti in forma normale.
- _ghcFlags mantiene la lista di flags utilizzati in fase di compilazione da GHC.
- _baseline specifica se il sistema deve includere nei tempi di benchmarking, anche i tempi impiegati a valutare in forma normale il risultato prodotto dalle procedure testate.

Per una documentazione più approfondita riguardo AutoBench ed il suo funzionamento invitiamo il lettore al paper [HH18].

1.4 Una conclusione errata

Ancor prima di dedicarci ai benchmark, è necessario scegliere l'implementazione funzionale di Quicksort con cui eseguire il confronto. A tal proposito, abbiamo scelto l'implementazione più comune sul web reperibile direttamente dall'Haskell Wiki [Pag20a].

```
quicksort :: (Ord a) => [a] -> [a]
quicksort xs = order xs where
  order (p:xs) = (order l) ++ [p] ++ (order r) where
    l = filter (<=p) xs
    r = filter (>p)  xs
  order xs = xs
```

Listing 1.7: Implementazione Haskell naive di Quicksort

Una seconda difficoltà sta nel comparare due funzioni scritte in differenti linguaggi: Haskell fornisce, attraverso l'interfaccia FFI (Foreign functions interface [Wik20]), una API con cui richiamare ed eseguire porzioni di codice compilate esterne al suo ecosistema. Il nostro interesse riguarda quindi la generazione ed il passaggio di una lista di *Int* da Haskell alla procedura C. Per fare ciò, è necessario allocare nuova memoria, copiare i valori dalla lista generata tramite QuickCheck, invocare la funzione ed infine eseguire nuovamente una conversione che riporti l'array ad una struttura dati equivalente in Haskell (nel nostro caso una *List*).

```
foreign import ccall unsafe "quickSort"
  quicksort_c :: Ptr CInt -> CInt -> CInt -> IO (Ptr CInt)
quickSortC :: [CInt] -> [CInt]
quickSortC xs = unsafePerformIO (withArrayLen xs \n p -> do
    quicksort_c p 0 (fromIntegral (n-1));
    peekArray n p)
```

Listing 1.8: Integrazione dell'implementazione Quicksort C tramite l'API FFI

Tali operazioni, introducono un overhead ad ogni chiamata del wrapper, il quale, oltre ad eseguire la funzione importata, si occupa di allocare, copiare ed infine deallocare la memoria, rendendo le misurazioni imprecise. Di conseguenza, l'approccio utilizzato è stato quello di testare tramite la libreria open source google/benchmark (per maggiori informazioni rimandiamo al repository GitHub google/benchmark [Ben20]) l'implementazione C di Quicksort, mentre per la versione Haskell, è stato utilizzato AutoBench.

1.4.1 Prepariamo AutoBench

Iniziamo col definire il tipo di istanze di input che gli algoritmi da testare utilizzeranno. Nel nostro caso, volendo ordinare delle liste, siamo interessati a delle sequenze appositamente non ordinate, su cui applicare le nostre funzioni. Inizialmente, definiamo il tipo di dato in input ai test attraverso la parola chiave *newtype*.

```
newtype RandomIntList = RandomIntList [Int] deriving Show
```

Listing 1.9: Definizione del tipo *RandomIntList*

Abbiamo quindi definito il tipo *RandomIntList* che racchiude una lista di *Int* e ne deriva i metodi per la visualizzazione da *Show*. A questo punto, definiamo ed implementiamo un'istanza *Arbitrary*, per il tipo *RandomIntList*. Ottenuto il parametro n in input, il metodo *arbitrary* invoca la funzione *sized* a cui viene associata una funzione lambda, in cui, tramite il parametro n, viene generata una lista casuale di lunghezza n incapsulata all'interno del nostro nuovo tipo *RandomIntList*.

```
instance Arbitrary RandomIntList where
  arbitrary = sized $ \n -> RandomIntList <$> vectorOf n
   arbitrary
```

Listing 1.10: Definizione del generatore di input per i test

AutoBench si occupa di ridurre in forma normale i parametri in input prima di eseguire le misurazioni, in modo tale da misurare solo il tempo di esecuzione, più quello necessario all'espansione del risultato. Per consentire ad AutoBench di ridurre *RandomIntList* in forma normale è necessario definire un istanza *NFData*.

```
instance NFData RandomIntList where
  rnf (RandomIntList xs) = rnf xs
```

Listing 1.11: Istanza di *NFData* per il tipo *RandomIntList*

Il metodo *rnf* si occupa di valutare il proprio parametro in forma normale. In questo caso, valutare in forma normale *RandomIntList*, equivale a ridurre la lista di interi associata.

```
test :: RandomIntList -> [Int]
test (RandomIntList xs) = func xs where
  func xs = ...
```

Listing 1.12: Definizione del wrapper per le funzioni da testare

Successivamente, incapsuliamo l'algoritmo di ordinamento all'interno di nuove procedure con un parametro di tipo *RandomIntList*.

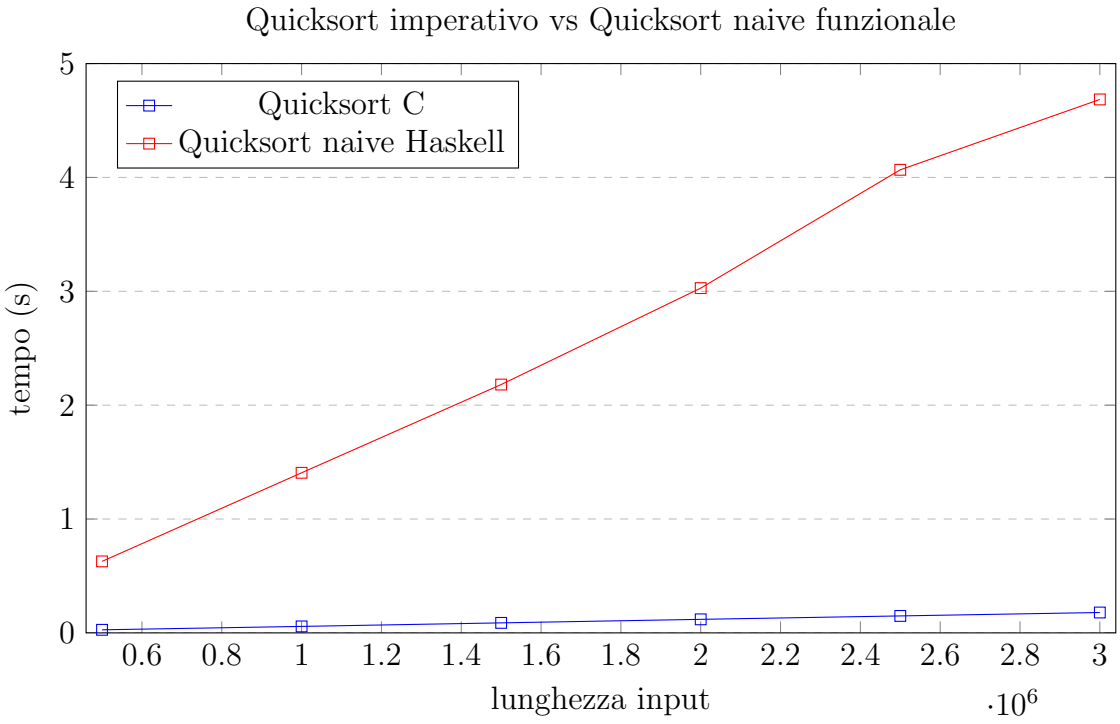
```
increasingTest :: TestSuite
increasingTest = def {
  _progs = ["quicksort"],
  _analOpts = def {
    _graphFP = Just "./graph.png",
    _reportFP = Just "./performance.txt",
    _coordsFP = Just "./csvdata.csv"
  },
  _dataOpts = Gen 0 500000 10000000
}
```

Listing 1.13: Definizione del TestSuite da eseguire

Non resta che definire i test suites da eseguire: siamo interessati a specificare le funzioni da eseguire, i path dei documenti in cui scrivere i report ed il generatore di n (dimensioni degli input) da cui costruire le liste. Definito il test suite ed inclusi i moduli necessari ad Autobench, siamo pronti ad eseguire i benchmark.

1.4.2 I primi risultati

I test sono stati condotti su una macchina con sistema operativo Ubuntu 20.04 LTS equipaggiata con un i9 9900K (con cpu scaling disabilitato durante i test), 16gb@3600mhz di ram ed ssd Samsung 860 EVO. Le istanze di input utilizzate, come mostrato in precedenza, sono di lunghezza crescente $0 \leq n \leq 3 \cdot 10^3$ ed incremento ad ogni test pari a $3 \cdot 10^6$. I risultati mostrano che tra le due implementazioni, la versione C, domina con un'efficienza 25 volte superiore. Rispondendo avventatamente alla nostra domanda, potremmo dire che Haskell, se da una parte consente di scrivere codice elegante e compatto, dall'altra cede gran parte dell'efficienza ottenibile con un programma C. A questo punto però ci domandiamo: è possibile fare di meglio? Quali sono le differenze che degradano così tanto le prestazioni nel caso funzionale? È interessante rispondere a queste domande poiché, come vedremo, porteranno alla luce alcune differenze chiave tra Haskell e C e consentiranno al lettore di comprendere più a fondo il funzionamento del linguaggio.



1.5 Game over, ritenta

I test eseguiti riportano una considerevole differenza prestazionale: possiamo fare sicuramente di meglio. Analizzando l'algoritmo di sort implementato in Data.List, notiamo un dettaglio molto interessante: l'algoritmo di ordinamento utilizzato è una versione funzionale bottom-up di Mergesort. A partire dal 2002, quest'ultimo, ha sostituito Quicksort subendo, diverse modifiche nel corso degli anni. Utilizzare Mergesort in un contesto funzionale può sembrare, a prima vista, limitativo in termini di performance: in realtà questo, si adatta molto meglio al caso delle liste. Haskell fornisce diverse strutture dati con cui maneggiare sequenze di oggetti, tra cui: le liste, gli array ed i vettori. La lista (*List*) è la struttura dati meglio integrata in Haskell. Le liste sono strutture coinduttive, implementate da una LinkedList, che offrono in $O(1)$ l'operatore cons (:) il quale pone un nuovo elemento in testa alla sequenza. La libreria standard Prelude, offre una serie di funzioni Higher Order con cui maneggiare liste: $foldr, map, filter$ sono solo alcuni esempi. Le liste sfruttano appieno la valutazione lazy: offrono un'interfaccia simile agli Iterator presenti in C++. Questo è il motivo per cui il programma riportato di seguito, è perfettamente contemplabile in Haskell: manipola delle strutture dati infinite attraverso la laziness.

```
even = [2,4..]
odd = [1,3..]
product = zipWith (*) even odd
```

Listing 1.14: Esempio di laziness in Haskell

D'altra parte, indicizzare un elemento in una lista, attraverso la funzione $indexOf$, richiede $O(k)$ dove k è l'indice richiesto. Le liste detengono, inoltre, una bassa data locality: le cpu sono soggette ad elevati fattori costanti dovuti al flusso continuo di dati tra cache e memoria principale. Al contrario, gli Array sono allocati in blocchi di memoria contigua e forniscono un'interfaccia con cui eseguire le operazioni

di lettura e scrittura in $O(1)$. Esistono svariate implementazioni di array tra cui IArray (immutabili) e Array.IO (array mutabili sincronizzati attraverso monad IO). La loro interfaccia è spesso complicata, da integrare con una porzione di codice funzionale e di conseguenza, il loro utilizzo è limitato a situazioni in cui le performance sono fondamentali. Inoltre, nel caso di array mutabili, il loro funzionamento non rispecchia il paradigma funzionale; possono cioè modificare il loro stato durante l'esecuzione. Negli ultimi anni, con l'intento di raffinare l'interfaccia di Array, Haskell ha introdotto la nuova API Vector: essa mantiene tutte le proprietà di un Array con un interfaccia più integrabile e semplice da utilizzare. Utilizzando le liste, Quicksort si scontra principalmente con 3 colli di bottiglia: la scelta del pivot è limitata al primo elemento in testa, ad ogni passo ricorsivo esegue 2 concatenazioni su liste e durante il partizionamento scorre la lista esattamente due volte. Sebbene la soluzione più semplice suggerisca l'uso di un array o vector, utilizzeremo le liste liste poiché, come abbiamo detto, sono maggiormente integrate e rispecchiano il vero paradigma funzionale su cui si basa Haskell. Nelle prossime sottosezioni, seguendo le idee presentate in [NHGW18], andremo a risolvere i problemi che limitano le performance della versione funzionale naive di Quicksort.

1.5.1 Il problema del partizionamento

Il primo problema da risolvere, riguarda il partizionamento: a differenza della versione imperativa, per ottenere le due sottoliste, stando alle condizioni sopracitate, è necessario un duplice scorrimento sull'intera lista. Il problema viene risolto attraverso la procedura di divisione *divide* con due liste di accumulazione l, r.

```
divide :: Int -> [Int] -> ([Int], [Int])
divide p xs = go p xs [] [] where
  go p [] l r = (l,r)
  go p (x:xs) l r
    | x < p = go p xs (x:l) r
    | x >= p = go p xs l (x:r)
```

Listing 1.15: Implementazione Haskell della procedura di partizione a singola passata

Dati il pivot e la lista, la procedura ricorsiva *divide* popola le due sottoliste l ed r, restituendole infine all'interno di una tupla binaria. Nonostante la procedura sfrutti la ricorsione, in questo caso è possibile utilizzare la tecnica di ottimizzazione di tail recursion. In questo caso, è sufficiente mantenere solo i parametri più recenti in memoria e non l'intero stack di chiamate.

1.5.2 Concatenazione tra liste: elegante ma inefficiente

Affrontiamo ora il problema più influente sulle performance della procedura Quicksort naive: le operazioni di concatenazione. Come abbiamo accennato, Haskell implementa le liste secondo una logica ricorsiva. Una conseguenza immediata è il fatto che per accedere all'elemento in coda alla lista xs sono necessari esattamente n passi ricorsivi dove $n = length\ xs$. Data l'espressione $xs + +ys$, l'operatore di concatenazione $(++)$ si occupa essenzialmente di inizializzare il puntatore dell'ultimo elemento di xs, all'elemento in testa nella seconda lista ys. Nonostante la sua complessità sia lineare alla lunghezza di xs, questa operazione svolge un ruolo chiave nella nostra funzione. Una soluzione potrebbe in realtà celarsi nell'albero di computazione ricorsivo eseguito

da Quicksort: sappiamo che dopo circa $O(nlog(n))$ passi ricorsivi, scegliendo il ramo sinistro (la sottolista dove ogni elemento $e \leq p$), la funzione identifica il primo elemento della lista ordinata.

```
smartQuicksort :: [Int] -> [Int]
smartQuicksort xs =
  let go [] a = a
      go (p:ps) a =
        let (leq, gt) = divide p ps
        in go leq (p:(go gt a))
  in go xs []
```

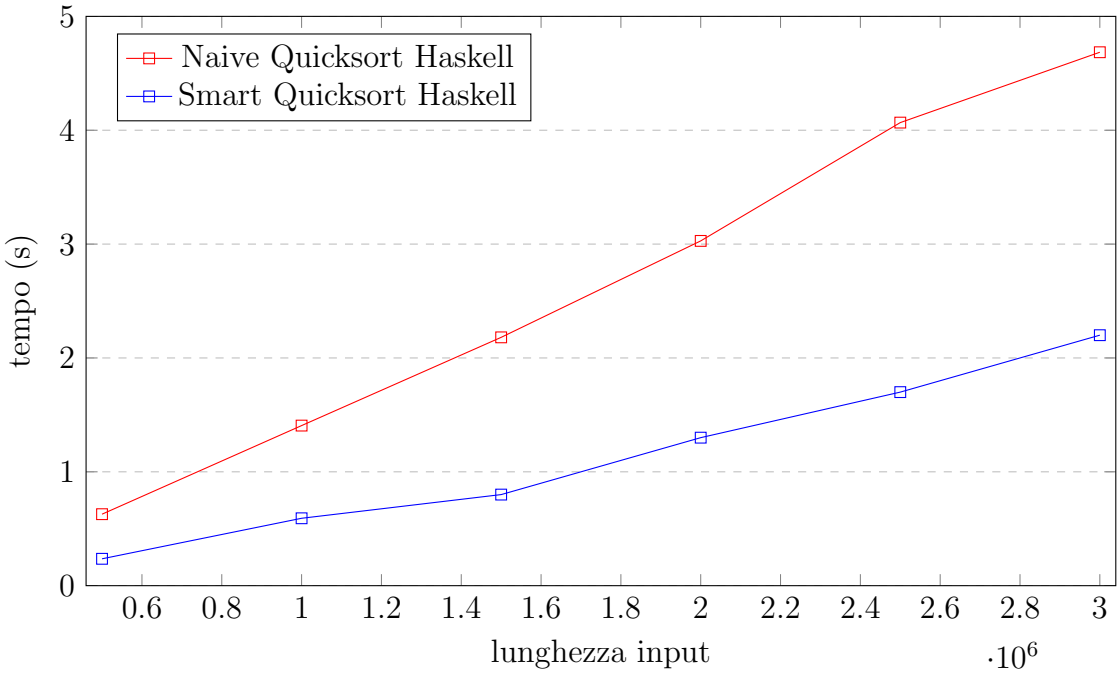
Listing 1.16: Implementazione Haskell di Quicksort smart

Nelle successive chiamate ai rami destri, durante la risalita, la procedura consumerà dapprima l'intero sotto-albero sinistro. Possiamo sfruttare questa proprietà, per popolare tramite l'operatore cons (:), la lista ordinata senza eseguire concatenazioni, ma semplicemente aggiungendo il pivot attuale alla testa della lista restituita dalla ricorsione sul ramo destro. Il parametro *a* della procedura *go*, agisce da accumulatore che ad ogni chiamata ricorsiva è uguale alla chiamata ricorsiva sul ramo destro con il pivot attuale aggiunto in testa. A questo punto non ci resta che ripetere i test con la nuova procedura per verificare l'efficacia delle modifiche apportate.

1.5.3 Test finale

A prima vista notiamo subito un calo marcato dei tempi di esecuzione. Nonostante l'eliminazione dell'operazione lineare di concatenazione (++) e l'ottimizzazione della procedura *divide*, le prestazioni non sono ancora comparabili alla versione imperativa in C. Scegliere un pivot casuale in tempo costante, rimane di fatto la proprietà chiave per ottenere una complessità al caso medio di $O(nlog(n))$.

Quicksort naive vs Quicksort smart in Haskell



Spesso siamo abituati alla scelta di un algoritmo valutando solo la sua complessità asintotica. Dai test eseguiti, è chiaro che il processo di valutazione di un algoritmo, prima di procedere alla sua implementazione, deve considerare anche il paradigma di programmazione utilizzato e le strutture dati disponibili. Inoltre, è fondamentale tenere a mente il numero di riduzioni intermedie necessarie e la possibilità di saturare la memoria se la laziness non venisse gestita nel modo corretto.

Capitolo 2

Il calcolo dei numeri primi

I numeri primi giocano oggi un ruolo più importante che mai, sebbene la loro comprensione è da considerarsi ancora ridotta. Molteplici rami dell'informatica e dell'ingegneria usufruiscono dei numeri primi: la crittografia, ad esempio, basa alcuni dei suoi teoremi di indecifrabilità sull'impossibilità di fattorizzare in fattori primi, numeri molto grandi (il sistema crittografico a chiave pubblica RSA utilizza numeri primi nell'ordine di 4096bit). Proprio per l'esistenza e validità di questi teoremi, generare numeri primi in modo efficiente, è tutt'ora una sfida aperta. In questo secondo capitolo, introdurremo il lettore al problema del calcolo dei numeri primi, riportando alcuni interessanti risultati ottenuti attraverso il "Teorema dei Numeri Primi". Presenteremo a seguito gli algoritmi, presi in analisi in questo studio, per la generazione di infiniti flussi di numeri primi. Partendo dalle idee alla base degli algoritmi, descriveremo il loro funzionamento, forniremo le implementazioni in Haskell ed infine ne deriveremo un limite superiore alla sua complessità computazionale.

2.1 I numeri primi

Un numero primo, è un numero diverso da 1 che non è il risultato del prodotto di due numeri naturali. Diremo quindi che un numero k soddisfa la proprietà di primalità se e solo se $k \in P$ dove P è l'insieme di tutti i numeri primi. Una prima interessante analisi riguarda la cardinalità dell'insieme P: il numero di elementi è finito? All'interno della sua raccolta Elements, Euclide prova formalmente, attraverso una dimostrazione per assurdo, l'infinità dell'insieme dei numeri primi P.

Teorema 1 (Teorema di Euclide)**.** *Dato un qualsiasi insieme di numeri primi P finito, esisterà sempre un $p \notin P$. Ne consegue che $|P| = \infty$.*

Dimostrazione. Consideriamo l'insieme finito P composto dai primi $p_1, p_2, ..., p_n$ e definiamo $Q = p_1 \cdot p_2 \cdot ... \cdot p_n$, $V = Q + 1$. Abbiamo ora due possibilità: se V è primo, allora esiste un primo non presente nella lista finita; altrimenti, se V non è primo, esisterà un fattore primo p che lo divide; se p fosse nella nostra lista, allora $p \mid V$ e $p \mid Q$; se p divide V e Q allora p dividerà anche $Q - V = V + 1 - V = 1$; per definizione non esiste alcun numero primo $k \mid 1$, perciò $p \notin P$. □

D'altronde, possiamo pensare al nostro insieme P come il generatore di $\mathbb{N}$ dove: $\forall n \in \mathbb{N}\ \exists!\ F = \{p_1, ..., p_k\} \subset P \mid n = p_1 \cdot ... \cdot p_k$. Definiremo inoltre composti tutti

quegli $n \in \mathbb{N}$ con $|F| > 1$, ovvero, tutti i numeri in $\mathbb{N}$ generati a partire da due o più moltiplicazioni tra numeri primi. L'infinità dei primi e quindi la possibilità di scegliere numeri arbitrariamente grandi, è una delle motivazioni per cui i sistemi crittografici moderni sono considerati sicuri: essi utilizzano numeri primi nell'ordine di migliaia di cifre per generare un hash crittografico con cui codificare e decodificare messaggi.

2.1.1 Il teorema dei numeri primi

Un seconda, interessante, considerazione, riguarda la distribuzione di P su $\mathbb{N}$. Come vedremo, approssimare la distribuzione con cui i numeri primi si presentano in *mathbbN* permetterà di fornire dei limiti superiori al numero di operazioni eseguite dagli algoritmi generatori di primi che presenteremo a seguito. Il teorema dei numeri primi (TNP) descrive la distribuzione asintotica dei primi su $\mathbb{N}$. Essenzialmente, formalizza l'idea che, al loro crescere, i primi diventano meno comuni, fornendo il tasso con cui ciò accade. Il teorema, sfruttando alcune idee introdotte da Bernhard Riemann, fornisce un approssimazione per la funzione di conteggio dei primi minori o uguali ad n ovvero $\pi(n)$.

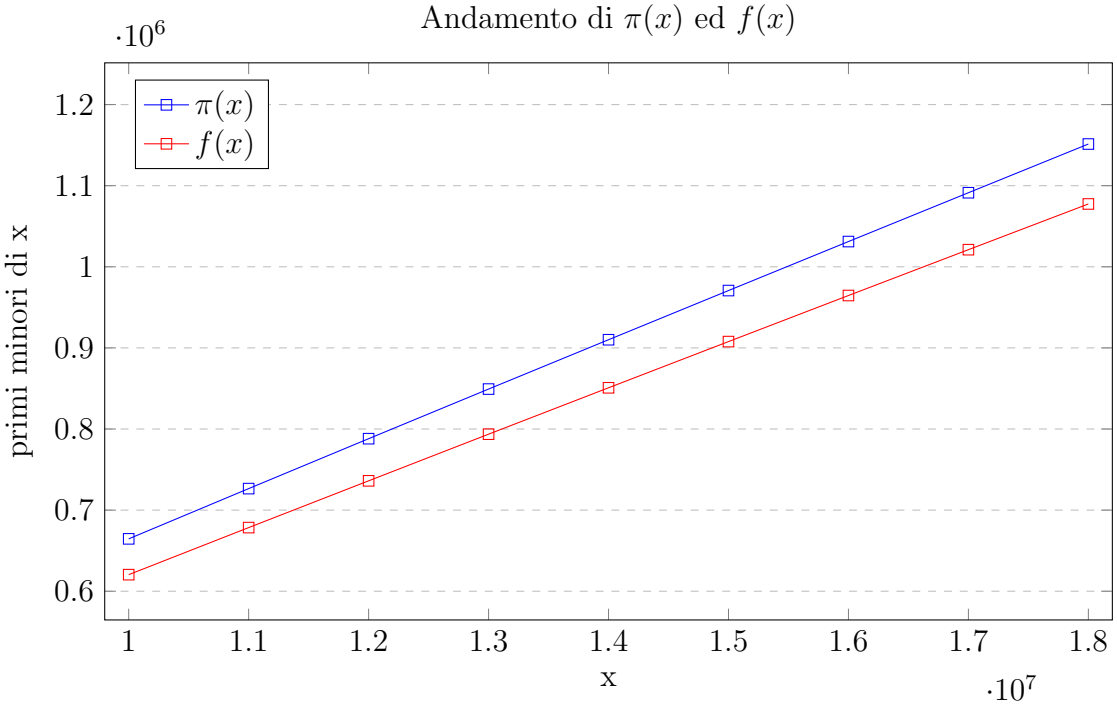
Teorema 1 (Teorema dei numeri primi)**.** *Definiamo $\pi(n)$ come la funzione di conteggio dei numeri primi. Allora la funzione $f(x) = \frac{x}{ln(x)}$ è una buona approssimazione per $\pi(x)$, ovvero il limite del quoziente delle due funzioni al crescere di x tende ad 1.*

$$\lim_{x\to\infty} \frac{\pi(x)}{\frac{x}{ln(x)}} = 1$$

Di conseguenza $\pi(x) \approx \frac{x}{ln(x)}$, ovvero l'errore relativo di $\pi(x) - f(x)$ tende a 0 al crescere di x.

Diamo ora un significato concreto ad $f(x)$: su un insieme di x numeri, la probabilità di incorrere in un numero primo è $\frac{1}{ln(x)}$. Di conseguenza la distanza media tra due numeri primi consecutivi p_i e p_{i+1} è circa $ln(p_i)$. Sommando k volte i prime gap, possiamo stabilire che il numero primo k-esimo è circa $p_k \approx kln(k)$.

Andamento di $\pi(x)$ ed $f(x)$



2.1.2 Metodi per il calcolo dei primi

Ancor prima di addentrarci nella descrizione dei metodi con cui calcolare sequenze di primi, è necessario definire più precisamente la sequenza che stiamo cercando di ottenere. Nel nostro caso siamo interessati a produrre la sequenza n-esima di primi $S_n = [2, 3..., p_n]$ in ordine crescente a partire da $p_1 = 2$ tale per cui presi due elementi consecutivi nella lista p_k, p_{k+1} non esiste alcun primo p_j con $k < j < k + 1$. I metodi di calcolo si dividono in due gruppi distinti, generativi e filtering: mentre con gli algoritmi generativi l'idea è quella di utilizzare una funzione matematica f con cui calcolare esattamente ciascun primo in $S_n = [f(1), f(2), ..., f(n)]$), nel caso del filtering si parte da una lista di interi $[2..n]$ rappresentante la porzione d'interesse dell'insieme $\mathbb{N}$ da cui, dopo aver eliminato tutti i composti, viene derivata la sequenza S_n. Sebbene gli algoritmi funzionali che presenteremo saranno tutti basati su filtering, forniremo una breve introduzione sui metodi generativi e sul perché risultano impraticabili.

È possibile generare numeri primi?

Per risolvere il problema secondo un approccio generativo, abbiamo bisogno di una funzione matematica $f : \mathbb{N} \to P$ definita come una combinazione lineare di funzioni elementari, che soddisfi le seguenti proprietà: $\forall m, n \in \mathbb{N}_{>0}$, $f(n) = p_n$ ovvero $f(n)$ è la funzione generatrice dell'n-esimo primo e se $n \neq m$ allora $f(m) \neq f(n)$ e $\forall p \in P \ \exists k \in \mathbb{N} \mid f(k) = p$. Sfortunatamente, una funzione con tali proprietà è inesistente per il seguente risultato sul campo dei numeri interi positivi: $\forall f \in \mathbb{Z}^k$ con $k > 0$ esistono infinite k-tuple tali che $f(x_1, ..., x_k)$ non sia primo (vedi [Rib97]). Riportiamo però l'esempio di $\lfloor p(x) \rfloor$, una funzione che soddisfa, in un intervallo finito di interi, le proprietà richieste ad f.

$$
\begin{aligned}
p(1) &= \sum_{i=1}^{\infty} \frac{p_i - 1}{\prod_{j=1}^{i} p_j} \\
p(x) &= \lfloor p(x-1) \rfloor (p(x-1) - \lfloor p(x-1) \rfloor + 1)
\end{aligned}
\tag{2.1}
$$

La funzione $\lfloor p(x) \rfloor$ permette il calcolo di un numero finito di primi attraverso l'espansione dei termini della serie in $p(1)$. Il limite di questa funzione è che il numero di primi calcolabili è strettamente correlato alla precisione del termine $p(1)$: utilizzando i primi 100 primi abbiamo $p(1) = 2.92005097731613471209256 29$. Nonostante ciò, tramite il termine $p(1)$ la funzione calcola correttamente solo i primi 18 numeri primi, richiedendo a priori la conoscenza degli altri 82 primi. Questo rende la funzione impraticabile nel nostro caso, seppur interessante da un punto di vista accademico.

Se non puoi generarli perché non filtrarli?

Una tecnica di filtering, verifica una proprietà ben definita su un insieme di elementi da filtrare, mantenendo solo gli elementi che la soddisfano. Calcolare la sequenza S_n tramite filtering è l'idea alla base del funzionamento di tutti gli algoritmi funzionali che presenteremo per generare numeri primi. I metodi con cui filtrare i primi sono molteplici: dalle semplici divisioni successive (inefficienti e poco scalabili) ai crivelli di Eratostene ed Eulero, fino ai test di primalità. Le tecniche di sieving si basano sul

concetto di filtrare, attraverso una conoscenza pregressa di un insieme di primi G, tutti i composti ottenuti attraverso la manipolazione di G producendo, a sua volta, nuovi primi con cui ripetere la procedura. Una proprietà interessante degli algoritmi di sieving è quella di calcolare i primi senza usufruire delle operazioni aritmetiche più costose, ovvero, quelle di divisione. Evitare tali operazioni o comunque ridurre il loro utilizzo influisce sulle performance finali dell'algoritmo. Un secondo metodo di filtering, applica un test di primalità ad ogni numero nel range $[2..n]$. Un test di primalità di occupa di verificare se n è primo attraverso un numero finito di passi. Per secoli, i matematici hanno tentato di definire un algoritmo polinomiale, deterministico, generale la cui correttezza potesse essere dimostrata attraverso teoremi già validati. Proprio per questo la maggior parte dei test di primalità valuta il numero fornito attraverso una procedura euristica: in questo modo è possibile ottenere dei tempi polinomiali e, in alcuni casi, una procedura generale applicabile ad ogni numero. Nel 2002, il test di primalità AKS, ha dimostrato che $PRIMES \in P$ fornendo un algoritmo tale da soddisfare le proprietà appena elencate. Ciononostante, le performance pratiche, seppure polinomiali, rendono l'algoritmo poco praticabile ma piuttosto un risultato fondamentale nell'informatica teorica.

2.1.3 Algoritmi imperativi di riferimento

Nonostante il nostro interesse sia rivolto a delle procedure in grado di generare sequenze infinite di primi in ambiente funzionale, molte delle idee per il calcolo dei primi sono già state implementate efficientemente attraverso una logica imperativa. In questo breve estratto introduciamo il lettore al crivello di Eratostene ed una sua versione più efficiente nel calcolo dei composti, il crivello di Eulero. Essi rappresentano i due metodi principe con cui ottenere la sequenza S_n a partire da un intervallo finito $[2..n]$. Entrambi gli algoritmi, applicano una tecnica di sieving, ovvero, escludono attraverso più passi di computazione tutti i numeri composti presenti nell'intervallo di interesse. Forniremo delle stime della complessità computazionale per entrambi gli algoritmi, utili a valutare quanto le implementazioni funzionali siano efficienti se comparate con quelle imperative.

Algoritmo di Eratostene

L'algoritmo di Eratostene è, per definizione, un algoritmo puramente imperativo: la descrizione fornita da Eratostene introduce, in modo astratto, all'utilizzo di una memoria mutabile. A partire dall'intervallo $[2..n]$ l'algoritmo, ad ogni passo k-esimo, preleva l'elemento successivo p_k non marcato nella lista; elimina a partire da p_k^2 tutti i suoi multipli, e, ripete la procedura fintanto che il prossimo $p_k^2 < n + 1$. Dalla descrizione ad alto livello del crivello di Eratostene è chiaro sin da subito che l'algoritmo ha bisogno di una struttura dati con cui memorizzare efficientemente il range $[2..n]$ e marcare i singoli numeri composti in tempo costante $O(1)$. Il problema è brillantemente risolto attraverso l'uso di un array mutabile in cui ogni cella corrisponde ad uno dei valori nel range d'interesse. Parallelamente, c'è un secondo problema da affrontare che riguarda l'insieme dei numeri composti generati e marcati nell'array: raramente un numero composto possiede un solo fattore primo distinto, motivo per cui molti multipli di 3 sono condivisi anche con 2 (6 è multiplo di 3 ma anche di 2). Definiamo $f(x)$ come la funzione che restituisce il numero di fattori primi unici di x: ogni composto c nella lista $[2..n]$ verrà generato, e quindi

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Figura 2.1: Il risultato del crivello di Eratostene per $n = 100$

eliminato, esattamente $f(c)$ volte. Il punto di forza del crivello di Eratostene è che la funzione $f(x)$ cresce molto lentamente al crescere di x, infatti $f(x) < 5$ per $n < 10^{34}$. Quindi se da una parte l'algoritmo esegue una quantità di lavoro inutile nel marcare multipli già marcati in precedenza, dall'altra il numero di queste operazioni superflue è limitato dalla funzione f che come abbiamo appena mostrato mantiene un andamento quasi costante. Da un punto di vista computazionale, al passo k-esimo, l'algoritmo genera ed elimina dall'array $\frac{n}{p_k}$ multipli. La nostra implementazione introduce però due ottimizzazioni interessanti: l'eliminazione dei multipli del primo p_k è avviata a partire da p_k^2 ed è terminata quando $p_k \cdot a > n$ con $a \geq p_k$. Di conseguenza, ad ogni passo di eliminazione, l'algoritmo risparmia la generazione di $p_k - 2$ composti, cioè

$$\sum_{k=1}^{\pi(\sqrt{n})} p_k - 2 \approx \frac{n}{ln(n)} - 2\pi(\sqrt{n})$$

derivata dal fatto che la somma dei primi n primi è circa $\frac{x^2 ln(x)}{2}$ dove $x \approx \frac{x}{ln(x)}$.

```
bool * sieveOfEratosthenes (int n) {
    bool * primes = (bool*)malloc((n+1) * sizeof(bool));
    memset(primes, true, sizeof(bool) * (n+1));
    for(int p = 2; p*p <= n; p++) {
        if(!primes[p]) continue;
        for(int i = p*p; i <= n; i+=p)
            primes[i] = false;
    }
    return primes;
}
```

Listing 2.1: Implementazione C dell'algoritmo di Eratostene

Otteniamo quindi che il numero di passi totali eseguiti dalla nostra versione imperativa dell'algoritmo di Eratostene sono circa

$$n \sum_{k=1}^{\pi(\sqrt{n})} \frac{1}{p_k} \approx nln(ln(\sqrt{n})) - \frac{n}{ln(n)} - \frac{4n}{\sqrt{n}ln(n)}$$

Come già anticipato, nel corso di questo capitolo, introdurremo una versione funzionale dell'algoritmo di Eratostene seguendo l'idea ricorsiva di Richard Bird.

Algoritmo di Eulero

L'algoritmo di Eulero si occupa invece di generare ed eliminare i numeri composti nel range $[2..n]$ una sola volta. Al passo k-esimo è quindi necessario calcolare solo i composti del primo p_k coprimi ai $k-1$ primi già identificati. Per fare ciò, l'idea alla base dell'algoritmo, è quella di prelevare l'elemento successivo p_k non marcato nella lista, calcolare tutti i composti moltiplicando p_k per tutti i sopravvissuti maggiori o uguali a p_k, ovvero $p_k \cdot s \; \forall s \; | s \; \geq p_k \wedge p_k \cdot s < n+1$, marcare nell'array i composti generati e ripetere la procedura fintanto che $p_k^2 < n+1$.

```
bool* sieveOfEuler(int n) {
    bool * primes = (bool*)malloc((n+1) * sizeof(bool));
    int * succ = (int*)malloc((n+1) * sizeof(int));
    int * prev = (int*)malloc((n+1) * sizeof(int));
    memset(primes, true, n+1);
    memset(succ, 1, n+1);
    memset(prev, 1, n+1);
    primes[0] = false;
    primes[1] = false;
    int prime = 2, latest;
    while(prime*prime < n+1) {
        for(int i = prime; prime*i < n+1; i += succ[i]) {
            primes[prime*i] = false;
            latest = prime*i + succ[prime*i];
            prev[latest] = succ[prime*i] + prev[prime*i];
        }
        while(latest > prime*prime) {
            int d = prev[latest];
            latest -= d;
            succ[latest] = d;
        }
        prime += succ[prime];
    }
    free(succ);
    free(prev);
    return primes;
}
```

Listing 2.2: Implementazione C dell'algoritmo di Eulero usando una lista doppiamente concatenata rappresentata come array

Ad ogni passo k-esimo, non consideriamo nel calcolo dei composti tutti quei numeri che contengono tra i loro fattori primi uno o più primi scoperti nei $k-1$ passi precedenti. Di conseguenza, il prodotto tra il nuovo primo p_k ed i composti sopravvissuti, non genererà mai un composto già marcato precedentemente. Seguire l'idea di Eulero per implementare un algoritmo imperativo, porta subito a scontrarsi con una nuova difficoltà: il punto di forza di Eulero è quello di saltare tutti i numeri composti già generati, ma come possiamo tradurre in codice questa idea? Nell'implementazione dell'algoritmo di Eulero presentata, il problema viene brillantemente risolto attraverso l'uso di 2 array *succ* e *prev*: lo scopo di questi array è quello di mantenere per ogni numero non marcato, l'offset da sommare per raggiungere il proprio successore o predecessore non ancora marcato. Durante l'eliminazione dei composti per il primo p_k, solo l'array dei predecessori viene aggiornato, mantenendo nel frattempo, una variabile *latest* che indica l'ultimo indice a cui *prev* è stato modificato. Successivamente, attraverso una seconda passata, partendo da *latest*, aggiorniamo l'array dei successivi *succ*. Il motivo di questa doppia passata sta nel fatto che aggiornando *succ* durante la fase di eliminazione dei composti non saremmo in grado di generarli tutti: ad esempio, nel caso $p_1 = 2$, eliminando 4 e aggiornando l'array di successivi *succ* non saremmo più in grado di generare il composto 8. A differenza dell'algoritmo di Eratostene, modellare il comportamento dell'algoritmo di Eulero, è un processo particolarmente complesso che necessita dell'impiego di strutture dati aggiuntive. Nonostante ciò, sebbene per ogni composto sono necessarie 3 operazioni di scrittura a differenza della singola operazione, necessaria nel caso di Eratostene, definito C_{p_k} come l'insieme dei composti generati da p_k coprimi ai $k-1$ primi precedenti, l'algoritmo di Eulero ha una complessità computazionale di $O(n)$, con un numero di operazioni nel range $[2..n]$ approssimabili a

$$\sum_{k=1}^{\pi(\sqrt{n})} 3|C_{p_k}| = 3(n - \pi(n)) \approx 3(n - \frac{n}{ln(n)})$$

Infatti, ad ogni passo k-esimo il for e $while$ interno sono eseguiti tante volte quanti sono i multipli del primo p_k coprimi ai $k-1$ primi già identificati. Nel corso di questo capitolo analizzeremo svariate modellazioni funzionali dell'algoritmo di Eulero e vedremo come, a differenza di Eratostene, la sua implementazione naive è molto più semplice e naturale in Haskell.

2.2 Passiamo ad Haskell

In questa sezione presentiamo al lettore 12 algoritmi funzionali per il calcolo di sequenze infinite di numeri primi in Haskell. Gli algoritmi sono stati suddivisi in 3 sottogruppi: elementari, avanzati ed ottimizzati.

2.2.1 Algoritmi elementari

I 3 algoritmi che presentiamo, esemplificano un primo tentativo di calcolare una sequenza S_n infinita di primi in Haskell, dalla struttura elegante e compatta ma sovente dalle performance inefficienti. In questo lavoro, ne vengono presentati alcuni considerati milestone della programmazione funzionale. Ad esempio l'algoritmo proposto da David Turner nel suo manuale di SASL risalente al 1983. Inoltre, la

loro analisi computazionale evidenzia alcune interessanti caratteristiche riguardo il modello operativo, poco evidenti se analizzate con occhio distratto, su cui è interessante soffermarsi.

Algoritmo di Turner

Iniziamo riportando una delle implementazioni più eleganti e compatte per il calcolo di numeri primi in Haskell. L'algoritmo simula la tecnica delle divisioni successive eliminando tutti i numeri $k \in \mathbb{N} \mid \exists p \in P \; con \; p \mid k$.

```
primes = filterPrime [2..] where
  filterPrime (p:xs) =
    p : filterPrime [x | x <- xs, (mod x p) /= 0]
```

Listing 2.3: Implementazione Haskell dell'algoritmo di Turner

Il lavoro svolto è pero di gran lunga superiore: ogni primo p_k è confrontato esattamente $k-1$ volte, a differenza di $\sqrt{k}$ volte nella versione ottimizzata dell'algoritmo delle divisioni successive. Definito n l'ultimo intero della lista $L = [2..n]$ da cui filtrare i primi, il lavoro svolto sui primi è

$$\sum_{i=1}^{\pi(n)} k - 1 = \frac{\pi(n)(\pi(n)-1)}{2} \approx \frac{n^2}{2(ln(n))^2}$$

A questo è necessario sommare il lavoro svolto per rimuovere i composti: notiamo che i multipli di 2 sono eliminati attraverso un singolo controllo, i multipli di 3 tramite 2 e così via. In questo caso sovrastimiamo il lavoro svolto poiché alcuni multipli sono in comune tra più primi. Abbiamo quindi

$$\sum_{k=1}^{\pi(\sqrt{n})} k\frac{n}{p_k} \approx \frac{n}{2} + n\sum_{k=2}^{\frac{2\sqrt{n}}{ln(n)}} \frac{k}{kln(k)} \approx \frac{n}{2} + n\int_{k=2}^{\frac{2\sqrt{n}}{ln(n)}} \frac{1}{ln(k)}\,dk \approx \frac{2n\sqrt{n}}{(ln(n))^2} + O(n)$$

Di conseguenza abbiamo che la complessità computazionale dell'algoritmo di Turner è $O(\frac{4n\sqrt{n}+n^2}{2(ln(n))^2})$.

Algoritmo di Eulero naive

L'algoritmo di Eulero naive segue invece l'idea imperativa, modellando la generazione dei composti del primo p_k, coprimi ai $k-1$ primi già identificati, attraverso l'annidamento di complementazioni di stream: ad ogni passo k-esimo, lo stream viene filtrato da tutti i multipli generati dal prodotto tra il primo p_k e gli elementi appartenenti allo stream di sopravvissuti al passo $k-1$. Formalizzando l'algoritmo, ad ogni passo k-esimo abbiamo

$$\begin{aligned} E_k &= \{n | n \; mod \; p_i \neq 0 \; \forall i \in [1, k-1] \wedge n \; mod \; p_k = 0\} \\ S_k &= \{n | n \; mod \; p_i \neq 0 \; \forall i \in [1, k]\} \end{aligned} \tag{2.2}$$

ovvero, gli insiemi contenenti nel primo caso, tutti i multipli di p_k che non condividono alcun fattore primo con l'insieme dei $k-1$ primi identificati precedentemente e, nel

secondo, i numeri sopravvissuti al k-esimo passo di eliminazione. Ponendo $p_1 = 2$ definiamo induttivamente la struttura di questi insiemi come

$$\begin{aligned} E_0 &= \varnothing \\ S_0 &= \{n | n \in \mathbb{N}_{>1}\} \\ E_{k+1} &= p_{k+1} S_k \\ S_{k+1} &= S_k \setminus E_{k+1} \end{aligned} \tag{2.3}$$

da cui deriva automaticamente che $S_{k+1} \subseteq S_k$ e $E_k \cap E_{k+1} = \emptyset$. Di conseguenza, ogni numero composto viene eliminato esattamente una volta. Seguendo la documentazione di Haskell (vedi [nHw20]), l'algoritmo ha una complessità di $O(i^2)$ per il primo p_i. L'obiettivo di questa sezione, è quello di fornire un limite superiore più preciso analizzando il lavoro svolto sui numeri composti e successivamente sui primi.

```
minus xs@(x:txs) ys@(y:tys)
  | x < y = x : minus txs ys
  | x > y = minus xs tys
  | otherwise = minus txs tys

primes = eulerSieve [2..] where
  eulerSieve cs@(p:tcs) = p:eulerSieve (minus tcs (map (p*)
   cs))
```

Listing 2.4: Implementazione Haskell dell'algoritmo di Eulero naive

Definiamo $n = iln(i)$ la stima del primo p_i richiesto. L'algoritmo analizzerà quindi gli elementi presenti in $\mathbb{N}_{\geq 2 \wedge \leq p_i} \subset \mathbb{N}$. Da questo insieme, l'algoritmo dovrà rimuovere $|C(\mathbb{N}_{\geq 2 \wedge \leq p_i})| = n - \pi(n)$ composti e restituire $\pi(n)$ primi. Ad ogni nuovo primo p_k identificato, l'algoritmo entra in un nuovo passo ricorsivo in cui lo stream generato dalla k-esima chiamata a *minus* dipende dai $k-1$ stream già presenti. Un numero composto c verrà sottoposto ad un massimo di

$$\frac{\sqrt{c}}{ln(\sqrt{c})} = \frac{2\sqrt{c}}{ln(c)} = O\Big(\frac{\sqrt{c}}{ln(c)}\Big)$$

controlli. Possiamo quindi fornire un primo largo limite superiore considerando $c = n$ costante, ottenendo

$$O\Big((n - \pi(n))\frac{\sqrt{n}}{ln(n)}\Big)$$

Tentiamo ora di raffinare il nostro ragionamento: anche qui, come nell'algoritmo di Turner, tutti i composti eliminati al passo k-esimo, hanno già superato $k-1$ rimozioni. Formalizzando, la serie

$$\sum_{k=1}^{\pi(n)} \frac{kn}{p_k}$$

conta il numero di confronti eseguiti sui multipli generati da ciascun primo, contando più volte i multipli in comune, nell'intervallo $\mathbb{N}_{\geq 2 \wedge \leq n}$. Riscriviamo la serie considerando che $p_k \approx kln(k)$ ed eliminando la singolarità presente per $k = 1$

$$\sum_{k=1}^{\pi(n)} \frac{kn}{p_k} = \sum_{k=1}^{\frac{n}{ln(n)}} \frac{kn}{kln(k)} = \frac{n}{2} + n \sum_{k=2}^{\frac{n}{ln(n)}} \frac{1}{ln(k)}$$

da cui otteniamo

$$n \int_{k=2}^{\frac{n}{ln(n)}} \frac{1}{ln(k)} dk \approx \frac{n^2}{(ln(n))^2} + O(n)$$

Concentriamoci ora sulle operazioni svolte sui primi: per identificare il primo p_k è necessario effettuare $k-1$ operazioni di controllo tra p_k e gli stream già esistenti. Il numero di controlli cresce linearmente con k, per cui

$$\frac{\pi(n)(\pi(n)-1)}{2} \approx \frac{n^2}{2(ln(n))^2}$$

stima il numero di operazioni svolte dall'algoritmo sui primi. In conclusione, abbiamo che un limite alla complessità computazionale dell'algoritmo di Eulero naive è $O(\frac{3n^2}{2(ln(n))^2})$.

Algoritmo delle divisioni successive

Torniamo per un momento all'algoritmo di Turner: l'algoritmo esegue delle operazioni di filtering ad ogni nuovo primo p_k identificato, effettuando quindi una quantità di divisioni superiori a quelle realmente necessarie. Prendiamo ad esempio il caso in cui l'algoritmo di Turner deve verificare se p_{500} (3571) è primo: p_{500} deve sopravvivere a 499 operazioni di filtering, quando in realtà, solo le prime 17 operazioni sono realmente necessarie. L'algoritmo delle divisioni successive applica quindi questa ottimizzazione all'algoritmo di Turner e lo fa tramite una procedura molto elegante.

```
primes = 2 : [i | i <- [3..],
 and [rem i p > 0 | p <- takeWhile (\p -> p^2 <= i) primes]]
```

Listing 2.5: Implementazione Haskell dell'algoritmo delle divisioni successive

L'algoritmo mantiene nella lista $L = [3..n]$ tutti i numeri k che sopravvivono ai controlli di divisibilità con i primi $\pi(\sqrt{k})$ primi. Ovviamente, nel caso $k = 2, 3$ il numero di elementi da controllare è nullo. Formalmente, il numero di operazioni eseguite sui primi è quindi

$$\sum_{k=3}^{\pi(n)} \pi(\sqrt{k}) \approx 2 \sum_{k=3}^{\pi(n)} \frac{\sqrt{k}}{ln(k)} \approx 2 \int_{k=3}^{\frac{n}{ln(n)}} \frac{\sqrt{kln(k)}}{ln(kln(k))} dk \approx \frac{4}{3} \frac{n\sqrt{n}}{(ln(n))^2}$$

mentre il lavoro svolto sui composti rimane invariato rispetto al valore calcolato per l'algoritmo di Turner. In conclusione, un limite alla complessità computazionale per l'algoritmo delle divisioni successive è $O(\frac{10n\sqrt{n}}{3(ln(n))^2})$, decisamente inferiore rispetto all'algoritmo proposto da Turner.

2.2.2 Algoritmi avanzati

Riportiamo ora, 3 algoritmi avanzati che modellano le idee imperative di Eratostene ed Eulero. Nel caso di Eulero, forniremo prima una soluzione che sfrutta la sequenza di Hamming e poi, una tecnica basata sulla generazione induttiva degli insiemi dei sopravvissuti, per generare i composti una sola volta. Nel caso di Eratostene invece seguiremo l'algoritmo proposto da Richard Bird e pubblicato da Melissa O'Neill's in [O'N09].

Algoritmo di Bird-Eratostene

L'algoritmo di Eratostene elimina iterativamente dalla lista ordinata $L = [2..n]$ tutti i multipli dei primi fino ad ora identificati. Il primo p_{k+1} è quel numero maggiore di p_k sopravvissuto all'eliminazione dei multipli dei $k-1$ primi dapprima identificati. Implementare questo algoritmo in ambiente funzionale, utilizzando le liste, è un processo particolarmente intricato e complesso. Riprendiamo per un momento l'algoritmo di Eratostene imperativo: la complessità computazionale dell'algoritmo è resa possibile dall'utilizzo di array con tempi di lettura e scrittura costanti. Implementare Eratostene in modo efficiente in Haskell richiede quindi un approccio differente: la modellazione che proporremo a breve, considera l'operazione di sieving come una procedura di eliminazione tra due liste ordinate, dove una rappresenta L, e l'altra, i composti. Di conseguenza, il problema si riduce a trovare un metodo atto a generare i composti in ordine crescente. Richard Bird propone un'implementazione basata sull'equazione ricorsiva

$$P = \mathbb{N}_{\geq 2} \setminus C(P)$$

da cui, tramite riscrittura

$$P = \{2\} \cup \mathbb{N}_{\geq 3} \setminus \bigcup_{k=1}^{\infty} p_k \mathbb{N}_{\geq p_k}$$

otteniamo un'equazione direttamente traducibile ad un algoritmo funzionale. Tramite questa seconda equazione possiamo definire un'implementazione che associa ad ogni nuovo primo identificato, la sua lista di multipli, ed elimina ad ogni passo, l'elemento k-esimo solo se uguale all'elemento minimo tra le teste degli stream di multipli attualmente presenti. Nella versione imperativa, l'algoritmo di Eratostene elimina i composti del primo p_k marcando le posizioni ad intervalli di p_k celle a partire da p_k^2. Nella versione funzionale, non potendo modificare ed accedere casualmente in tempo costante ad una lista, è necessaria una procedura con cui ridurre la lista di liste di multipli ad un flusso ordinato e univoco di composti da eliminare, attraverso la funzione *minus*, da L.

```
union xs@(x:txs) ys@(y:tys)
  | x == y = x:union txs tys
  | x < y = x:union txs ys
  | x > y = y:union xs tys

primes = 2:([3..] `minus` comp) where
  comp = foldr unionP [] [multiples p | p <- primes]
  multiples n = map (n*) [n..]
  unionP (x:xs) ys = x:union xs ys
```

Listing 2.6: Implementazione Haskell dell'algoritmo di Bird-Eratostene

Tramite $foldr$, percorriamo la lista di liste ricorsivamente, applicando ad ogni passo ricorsivo la funzione *union*.

```
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
```

Listing 2.7: Implementazione Haskell di $foldr$

Perché mai, allora, utilizzare *unionP*? La risposta si trova nell'implementazione di $foldr$: al primo passo ricorsivo $foldr$ applica la funzione *union* alla prima lista di multipli, richiamando se stessa al secondo parametro. In questo modo la funzione *union* non sarà mai produttiva poiché non è in grado di eseguire il confronto con il valore y restituito dal secondo parametro. Sfruttando le proprietà

$$\begin{aligned} &\forall k > 0 \; p_{k-1}^2 < p_k^2 \\ &\exists p_j \mid p_k < p_j < p_k^2 \end{aligned} \tag{2.4}$$

la funzione *unionP*, inserisce in testa alla lista risultante il primo elemento x della lista xs, garantendo almeno un elemento con cui rendere *union* produttiva. Tornando alla complessità, la quantità di lavoro svolto da *union* per il k-esimo numero da filtrare è quindi lineare rispetto al numero di primi minori di $\sqrt{k}$

$$\sum_{i=1}^{\pi(\sqrt{k})} \frac{i}{p_i} = \int_{i=2}^{\frac{2\sqrt{k}}{ln(k)}} \frac{1}{ln(i)} di = \frac{\sqrt{k}}{(ln(k))^2}$$

Il costo di ogni singola operazione di eliminazione è quindi dipendente dal costo unitario dell'operazione di *union*. Di conseguenza, ad ogni operazione svolta nel caso dell'algoritmo imperativo, è necessario aggiungere un numero di operazioni pari a $\frac{\sqrt{n}}{(ln(n))^2}$ da cui deriva che, un limite alla complessità computazionale dell'algoritmo di Bird-Eratostene è $O(\frac{n\sqrt{n}ln(ln(n))}{(ln(n))^2})$. È interessante notare inoltre che l'algoritmo delle divisioni successive è più efficiente dell'algoritmo di Bird-Eratostene di un fattore $ln(ln(n))$: questo non introduce un peggioramento marcato, in quanto, il fattore aggiuntivo cresce molto lentamente. Nella pratica, l'algoritmo di Bird-Eratostene è praticabile per qualsiasi istanza di n calcolabile tramite divisioni successive e, nel caso di n ridotti, riporta dei tempi 4x più efficienti.

Algoritmo di Eulero-Hamming

Vediamo ora una seconda versione dell'algoritmo di Eulero, che fa uso della sequenza di Hamming. La sequenza di Hamming è l'insieme dei numeri regolari, ovvero i numeri appartenenti all'insieme $H(B) = \{n | n \in B^k \; con \; k > 0\}$ dove $B = \{2, 3, 5\}$. Una definizione equivalente dell'insieme $H(B)$ è data dall'insieme con cardinalità minima che soddisfa l'equazione

$$H(B) = b \cdot H(P) \cup H(B \setminus \{b\})$$

Come nella precedente versione di Eulero, siamo interessati a produrre l'insieme dei numeri composti generando ciascun composto una sola volta. Ponendo $B = P$, il nostro obiettivo è quindi quello di generare, a partire dalla base B, l'insieme dei numeri risultanti dal prodotto di 2 o più elementi appartenenti a B. Proprio per questo, siamo interessati all'insieme

$$H'(B) = H(B) \setminus \{1\}$$

che a sua volta traduce l'equazione introdotta poc'anzi in

$$H'(B) = \{b\} \cup b \cdot H'(B) \cup H'(B \setminus \{b\})$$

da cui notiamo immediatamente la proprietà che caratterizza l'algoritmo di Eulero: ponendo $B = P$ gli insiemi $p \cdot H'(P) \cap H'(P \setminus \{p\} = \varnothing$, poiché il primo insieme possiede solo i numeri con fattore primo p. Di conseguenza abbiamo che

$$\begin{aligned} H(P) &= \mathbb{N} \\ 2 \cdot H(P) &= \{2\} \cup E_1 \\ 3 \cdot H(P_1) &= \{3\} \cup E_2 \end{aligned} \tag{2.5}$$

ovvero, tutti i numeri pari sono eliminati alla prima iterazione, seguiti da tutti i multipli di 3 che però non sono multipli di 2 alla seconda iterazione, cioè i composti generati nelle prime due iterazioni dell'algoritmo di Eulero naive. Essendo interessati all'insieme

$$C(P) = H(P) \setminus P$$

è necessario evitare l'inserimento dei primi nella lista dei composti. D'altra parte, per calcolare correttamente l'insieme di composti a partire dal primo p_k, è necessario considerare anche i primi a partire da p_{k+1}, i quali però, non sono mai restituiti dalla chiamata ricorsiva a *comp*. Per ovviare a questo problema, viene eseguita l'unione tra *xs* e *cmpsts* prima di effettuare la chiamata ricorsiva.

```
sMinus xs@(x:txs) ys@(y:tys)
  | x == y = sMinus txs tys
  | otherwise = x:sMinus txs ys

dUnion xs@(x:txs) ys@(y:tys)
  | x < y = x:dUnion txs ys
  | otherwise = y:dUnion xs tys
dUnion xs [] = xs

primes = 2:([3..] `sMinus` comp primes) where
  comp (x:xs) = cmps where
    cmps = x*x:map (x*) (dUnion xs cmps) `dUnion` (comp xs)
```

Listing 2.8: Implementazione Haskell dell'algoritmo di Eulero-Hamming

Sfruttando a nostro vantaggio la proprietà che la lista di composti è strettamente maggiore di quella dei primi, possiamo ulteriormente ottimizzare la funzione *minus* escludendo il caso $x > y$. Per fornire un limite superiore alle operazioni eseguite dall'algoritmo, prendiamo in considerazione solo il lavoro svolto sui composti: dato un range $L = [2..n]$ in cui identificare i primi, dobbiamo quindi generare l'insieme $C(P)$ tale che $\mathbb{N}_{>1 \wedge \leq n} \setminus C(P) = P$. Notiamo innanzitutto che, per generare l'insieme dei composti $C(P)$ fino ad n, l'algoritmo richiama ricorsivamente *cmps* esattamente $\pi(\sqrt{n})$ e calcola per ogni primo p_k, $O(\frac{n}{p_k})$ composti, considerando del lavoro aggiuntivo per i multipli in comune con altri primi. Per estrapolare i composti dallo stream del primo k-esimo, l'algoritmo deve processare k *dUnion*. Abbiamo perciò

$$\sum_{k=1}^{\pi(\sqrt{n})} \frac{kn}{p_k} = \frac{n}{2} + n \sum_{k=2}^{\pi(\sqrt{n})} \frac{1}{ln(k)} = \frac{n}{2} + n \int_{k=2}^{\pi(\sqrt{n})} \frac{1}{ln(k)} dk \approx \frac{2n\sqrt{n}}{(ln(n))^2} + O(n)$$

da cui deriviamo che, un limite alla complessità computazionale dell'algoritmo di Eulero-Hamming è $O(\frac{2n\sqrt{n}}{(ln(n))^2})$.

Algoritmo di Eulero Bird-Salvo

L'implementazione dell'algoritmo di Eulero naive con cui generiamo gli insiemi E_k ed S_k prevede l'annidamento di complementazioni di stream attraverso la funzione *minus*. Interpretando l'equazione differentemente, possiamo generare l'insieme dei primi, calcolando l'insieme dei composti attraverso l'unione degli insiemi dei multipli per ogni primo, ovvero

$$P = \mathbb{N}_{\geq 2} \setminus \bigcup_{k=1}^{\pi(\sqrt{n})} E_k$$

Ad ogni passo k-esimo l'algoritmo identifica un nuovo primo e genera i due insiemi E_k, S_k induttivamente attraverso le equazioni riportate di seguito

$$\begin{aligned} E_{k+1} &= p_{k+1} S_k \\ S_{k+1} &= S_k \setminus E_{k+1} \end{aligned} \tag{2.6}$$

da cui ne deriva immediatamente l'implementazione funzionale.

```
primes = 2:([3..] `sMinus` (comp primes [2..])) where
  comp (p:ps) ss@(s:tss) = es `dUnionP` comp ps ss' where
    es = map (p*) ss
    ss' = tss `sMinus` es
    dUnionP (x:xs) ys = x : dUnion xs ys
```

Listing 2.9: Implementazione Haskell dell'algoritmo di Eulero Bird-Salvo

Ad ogni chiamata, *comp*, genera l'insieme S_{k+1} tramite l'operazione di *minus* tra tss ed E_{k+1}, ovvero es.

2.2.3 Tecniche di ottimizzazione

Introduciamo ora il lettore ad alcune tecniche con cui ridurre il numero di operazioni svolte dai nostri algoritmi a parità di primo p_k richiesto. Dapprima presenteremo la tecnica delle ruote con cui siamo in grado di ridurre la densità della lista $L = [2..n]$ considerando solo i numeri coprimi ad un insieme ristretto di primi. Successivamente, andremo ad occuparci del problema legato alle operazioni di stream fusion: per ovviare a tale problema, faremo uso di una coda di priorità, implementata attraverso un heap binomiale lazy, ovvero, una struttura dati organizzata secondo una foresta di alberi.

Ruote

In tutti gli algoritmi funzionali per il calcolo di numeri primi presentati fino ad ora, è necessario fornire una lista iniziale L, rappresentante $\mathbb{N}$, da cui rimuovere tutti i numeri composti. Un problema che pone un limite alle performance dei nostri algoritmi, è che al crescere di n, il prime gap medio tra i numeri primi adiacenti, cresce anch'esso di $ln(n)$ con la conseguenza che il numero di composti da rimuovere dalla lista, per ottenere il primo successivo, cresce costantemente. A questo, proponiamo una soluzione basata su ruote: l'idea è quella di generare la lista iniziale già filtrata dai multipli di un insieme finito di primi. L'idea parte da una semplice osservazione: eliminando tutti i multipli di 2, la densità di $\mathbb{N}$ si riduce del 50%, riducendo quindi il

lavoro svolto dall'algoritmo. In Haskell possiamo ottenere una lista di soli numeri dispari, attraverso la list comprehension $[3, 5..]$. Perché fermarci ai soli multipli di 2? Generando solo i numeri coprimi all'insieme di primi $R = \{2, 3, 5, 7\}$ circoscriveremmo la ricerca solo al 23% dei numeri originariamente presenti in $\mathbb{N}$. Una ruota è quindi una lista di offset, che indica, a partire dal primo p_{k+1}, quante posizioni debbono esser saltate per raggiungere il numero successivo coprimo ai $k = |R|$ primi precedenti. Ad esempio, nel caso di $p_{k+1} = 5$ avremo la ruota $[2, 4]$ che, a partire da 5, genera tutti i numeri coprimi ad $R = \{2, 3\}$, ovvero $5, 7, 11, 13, 17, 19....$ Riportiamo di seguito una funzione che, data una lista di primi xs, genera la ruota corrispondente.

```
genWheel :: [Int] -> [Int]
genWheel [2] = [1]
genWheel xs
  | length cpl == 0 = []
  | otherwise = go (head cpl) (tail cpl) where
    c = foldr (*) 1 xs
    cpl = [x | x <- [2..(c+1)], cp x]
    cp x = foldr (&&) True [mod x p /= 0 | p <- xs]
    go a (b:t) = (b - a) : go b t
    go a [] = [(head cpl + c) - a]
w3 = genWheel [2,3,5]
```

Listing 2.10: Implementazione Haskell dell'algoritmo della ruota w_k

In altre parole, sfruttiamo la periodicità dei multipli dei primi in xs per generare solo i numeri coprimi ad essi: non a caso, la struttura dati è denominata ruota. Scelti i primi k numeri primi, definiamo la circonferenza della ruota come $c = p_1 \cdot p_2 \cdot ... \cdot p_k$ da cui generiamo la lista di numeri nel range $0 < a < c$ con $mcd(a, c_k) = 1$, ovvero, tutti i numeri coprimi alla circonferenza e di conseguenza, coprimi ai k primi in R. Partendo da p_{k+1} generiamo, infine, la lista L_k delle distanze tra i coprimi identificati. Con il termine "girare la ruota", intendiamo il calcolo del numero coprimo a c_k, ottenuto sommando periodicamente al numero generato in precedenza, la corrispondente distanza in L_k.

```
circ w = w ++ circ w
spin (w:ws) n = n:spin ws (n+w)
coprimes = spin (circ w3) 7
```

Listing 2.11: Generazione dei coprimi tramite la ruota w_3 in Haskell

Per generare la lista di coprimi, facciamo uso delle funzioni *spin* e *circ*, le quali, si occupano rispettivamente di generare l'elemento successivo ed appendere L_k alla vecchia lista delle distanze oramai consumata da *spin*. É possibile, inoltre, generare a partire dalla ruota w_k la ruota w_{k+1} attraverso una semplice funzione Haskell che riportiamo qui di seguito. Notiamo immediatamente che p_{k+1}, ovvero il primo numero coprimo in w_k, è ora necessario al calcolo di c_{k+1}: per questo, il primo passo replica esattamente p_{k+1} volte L_k producendo L_{k+1}.

```
nextWheel [] _ _ = [1]
nextWheel (w:ws) p np = nWAux (rep p (ws++[w])) np p where
  nWAux [] _ _ = []
  nWAux [w] _ _ = [w]
  nWAux (w:ws) s p
    | mod (w+s) p == 0 = nWAux ((w+head ws):(tail ws)) s p
    | otherwise = w:nWAux ws (w+s) p
  rep 0 _ = []
  rep n xs = xs ++ rep (n-1) xs
```

Listing 2.12: Implementazione Haskell dell'algoritmo della ruota successiva

Successivamente, ponendo $d = p_{k+2}$ sommiamo iterativamente le distanze in $Lk + 1$ a d eseguendo la fusione della posizione attuale e successiva in L_{k+1}, se $d \mid p_{k+1}$.

$$\begin{aligned} & p_2 = 3; \quad c_2 = 2; \quad p_3 = 5; \quad c_3 = 6 \\ & L_2 = [2, 4]; \\ & L_3 = [2, 4, 2, 4, 2, 4, 2, 4, 2, 4]; \\ & L_3 = [7, 11, 13, 17, 19, 23, \mathbf{25}, 29, 31, \mathbf{35}] \\ & L_3 = [4, 2, 4, 2, 4, 2 + 4, 2, 4 + 2] \\ & L_3 = [4, 2, 4, 2, 4, 6, 2, 6] \end{aligned} \tag{2.7}$$

Da qui in poi utilizzeremo la dicitura $\mathbb{N}_{w_k}$, per indicare il sottoinsieme generato dalla ruota wk.

Heap binomiali lazy

In tutti gli algoritmi presentati, abbiamo modellato l'eliminazione dei numeri composti dalla lista iniziale L rappresentante $\mathbb{N}$, attraverso delle operazioni di stream fusion: durante queste operazioni, due o più stream vengono ridotti ad un nuovo stream. Nei nostri algoritmi questa operazione è molto frequente: ad esempio, l'algoritmo di Bird-Eratostene, esegue delle operazioni di unione tra un numero di liste pari a k, quando il primo richiesto è p_k; oppure nel caso dell'algoritmo di Eulero naive, in cui ogni *minus* è innestata all'interno delle *minus* precedenti, creando una serie di dipendenze tra stream. Queste operazioni introducono un limite alle performance dei nostri algoritmi: scegliere k elementi da m stream, non è un'operazione nell'ordine di k, bensì, essendo m variabile, nell'ordine di $O(km)$. Il lavoro che vogliamo ottimizzare, è quello dedicato all'identificazione dello stream xs con $head\ xs$ minimo: se questa operazione fosse eseguita in modo efficiente, allora saremmo anche in grado di eliminare gli elementi dalla lista di partenza $[2..n]$ in modo efficiente. Per mantenere degli stream ordinati, è necessaria una coda di priorità e la scelta di una chiave, in questo caso $head\ xs$, con cui ordinarli. La richiesta si traduce nell'identificare una struttura dati efficiente con cui implementare una coda di priorità: siamo interessati ad inserire ed estrarre nel minor tempo possibile (possibilmente entrambi in $O(1)$). Se esistesse una struttura dati con procedure di inserimento (con ordinamento all'interno della struttura dati) ed estrazione (con relativo ordinamento) in $O(1)$, allora, potremmo ordinare m elementi in $O(m)$: questo però ci porta ad una contraddizione con il limite inferiore $O(mlog(m))$ fornito dal teorema del limite inferiore per algoritmi

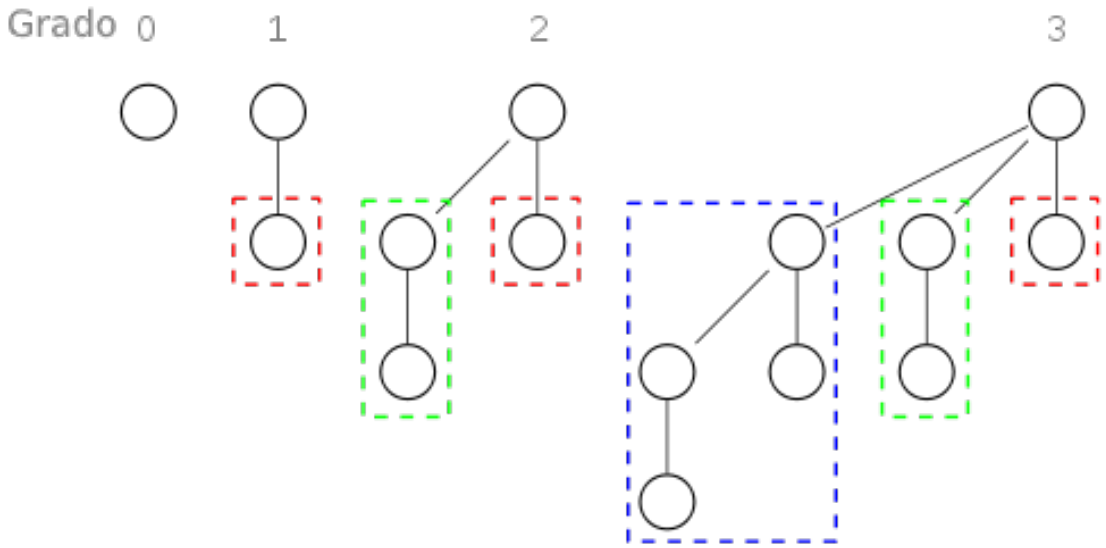


Figura 2.2: Esempio di alberi binomiali

basati su confronti. La struttura dati che più si avvicina alle nostre esigenze è l'heap binomiale lazy, una struttura composta da una foresta di alberi binomiali connessi tra loro, attraverso una lista a puntatori sulle radici. Un heap binomiale lazy, offre una procedura di inserimento in $O(1)$ ammortizzato, ricerca del minimo in $O(1)$ ed estrazione del minimo in $O(log(n))$ ammortizzato. La struttura dati alla base dell'heap binomiale è l'albero binomiale, ovvero, un albero con ordinamento tra nodi, definito ricorsivamente come segue: l'albero binomiale B_0 è composto da un solo nodo, mentre l'albero B_k è costituito da due alberi B_{k-1} connessi in modo che, la radice di uno, sia figlio sinistro della radice dell'altro. Dati n elementi, l'heap binomiale è composto da una lista $TREES$ con $k = log(n)$ alberi binomiali ordinati per grado g (numero di nodi) in cui, la proprietà $\forall t_1, t_2 \in TREES\ g(t_1) \neq g(t_2)$ è valida. Ad ogni inserimento, l'elemento viene inserito in un nuovo albero t_{new} con $g(t_{new}) = 1$ eseguendo la procedura di fusione in $O(log(n))$ nel caso in cui dovessero verificarsi dei conflitti tra gradi. Attraverso l'utilizzo di una tecnica lazy, possiamo ridurre la complessità d'inserimento a $O(1)$: rimandiamo la risoluzione dei conflitti tra gradi alla procedura di estrazione. Di conseguenza, durante la fase di estrazione del minimo, preleviamo in $O(t)$ (con $log(n) \leq t \leq n$) la radice minima dalla lista $TREES$, ed inseriamo se presenti, i due sottoalberi binomiali figli esposti. Successivamente, applichiamo una procedura di coalescenza, con cui andiamo a risolvere in $O(t + log(n))$ i conflitti tra gradi ristabilendo a $log(n)$ il numero di alberi binomiali attraverso i seguenti passi:

- Genera un array con $log(n)$ buckets ordinati per grado in $O(log(n))$
- Inserisce gli t alberi binomiali nei rispettivi bucket, seguendo il loro grado in $O(t)$
- Esegue la fusione degli alberi in $O(t)$.

Sebbene l'estrazione del minimo è ora eseguita in $O(n)$ nel caso peggiore, tramite l'analisi ammortizzata dei costi (in questo caso tramite il metodo potenziale) risulta banale verificare che il costo ammortizzato è $O(log(n))$. Negli algoritmi ottimizzati che presenteremo, la coda di priorità con heap binomiale lazy utilizzata, è Prio.min, appartenente al package pqueue. La coda di priorità Prio.min, offre la possibilità di inserire coppie chiave-valore e, possiede un'interfaccia facilmente integrabile con i nostri algoritmi (per un analisi più approfondita rimandiamo alla pagina Hackage di pqueue [Pag20b]).

2.2.4 Algoritmi ottimizzati

Riportiamo infine, 7 algoritmi ottimizzati attraverso l'uso di ruote e code di priorità. Seguendo le implementazioni presentate da Melissa O'Neill's in [O'N09], e, sfruttando le modifiche proposte dal professor Salvo in [SP18], illustreremo due versioni dell'algoritmo di Eulero con code di priorità e ruote. Infine, con lo scopo di filtrare a priori $\mathbb{N}$, equipaggeremo con una ruota w_4 gli algoritmi di Bird-Eratostene, Eulero Bird-Salvo, Eulero-Hamming e le due versioni già ottimizzate di Oneills-Eulero che vedremo qui di seguito.

Algoritmo di Oneills-Eulero

L'impiego di procedure come *union* e *minus* per fondere stream introduce, una complessità aggiuntiva nella gestione delle centinaia di stream generate durante l'esecuzione dell'algoritmo. Nel caso della versione dell'algoritmo di Eulero naive, il numero di stream per il primo i-esimo è pari ad $s(i) = 2i - 1$ mentre nell'algoritmo avanzato di Bird-Eratostene, abbiamo $s(i) = i + 3$, cioè, un numero decisamente inferiore seppur lineare all'indice del primo richiesto. Ridurre questa complessità lineare sulle operazione di fusione, è necessaria al fine di calcolare primi con indice nell'ordine di milioni, in tempi ridotti. In [O'N09] Melissa O'Neill's, descrive un'implementazione dell'algoritmo di Eulero in grado di eliminare completamente le operazioni di fusione, ottenendo un miglioramento delle sostanziale prestazioni. Presentiamo ora la versione in [SP18], proposta dal professor Salvo, seguendo l'idea di computazione basata su insiemi S_k, E_k. Il funzionamento dell'algoritmo, prevede quindi l'impiego di una coda di priorità (attraverso un heap binomiale minimo) i cui elementi sono coppie $(chiave,\ valore)$ ognuna delle quali associate al primo p_k. Definito lo stream dei composti per il primo k-esimo $xs = map\ (p_k*)\ [p_k..]$, la chiave sarà il primo elemento dello stream, ovvero, $head\ xs$ mentre il valore, lo stream non ancora valutato a partire dall'elemento successivo, ovvero $tail\ (map\ (p_k*)\ [p_k..])$. Ad ogni nuovo numero n, l'algoritmo preleverà quindi la coppia con chiave c_{min} minima, ovvero il composto minimo, classificando n primo se $c_{min} > n$ o composto altrimenti. Nel caso in cui n risultasse composto, l'algoritmo provvederà ad eliminare n dalla lista finale di primi, aggiornando la coppia $(c_{min},\ xs = map\ (p_k*)\ [c_{min} + 1..])$ con $(head\ xs,\ tail\ xs)$. È possibile combinare code di priorità e ruote, ottenendo un'implementazione più efficiente.

```
primes = sv [2..] where
  sieve (c:cs) =
    c:sieve' cs ss (insert (c*c) (tail es) empty) where
      es = map (c*) (c:cs)
      ss = cs `sMinus` es
      sieve' cs@(c:tcs) ss tbl
        | c < n = c : sieve' tcs ss' tbl'
        | otherwise = sieve' tcs ss tbl'' where
          (n, m:ms) = findMin tbl
          es = map (c*) ss
          ss' = tail (ss `sMinus` es)
          tbl' = insert (c*c) (tail es) tbl
          tbl'' = insert m ms (deleteMin tbl)
```

Listing 2.13: Implementazione Haskell dell'algoritmo di ONeills-Eulero con coda di priorità

All'identificazione del primo k-esimo, l'algoritmo genera w_k attraverso la funzione $nextWheel$ e la ruota w_{k-1}: lo stream di elementi confrontati con i composti, è quindi, ad ogni nuovo primo identificato, filtrato attraverso l'uso della ruota w_k. Al contempo, lo stream di composti per il primo p_k, è generato moltiplicando la ruota w_{k-1} per il nuovo primo p_k. In questo modo, riduciamo il numero di confronti necessari all'identificazione di nuovi primi ed occupiamo una quantità di memoria decisamente inferiore.

```
s4 = spin (circ w4) 11
primes = 2:3:5:7:sieve s4 w4 where
  sieve (c:cs) w = c:sieve' cs (nextWheel w c)
    (insert (c*c) (circ (map (c*) w)) empty) where
      sieve' cs@(c:tcs) w tbl
        | c < n = c : sieve' tcs w' tbl'
        | otherwise = sieve' tcs w tbl'' where
          (n, m:ms) = findMin tbl
          w' = nextWheel w c
          tbl' = insert (c*c) (circ (map (c*) w)) tbl
          tbl'' = insert (n+m) ms (deleteMin tbl)
```

Listing 2.14: Implementazione Haskell dell'algoritmo di Oneills-Eulero con coda di priorità e $nextWheel$

Filtrare $\mathbb{N}$ tramite la ruota w_4

Riprendiamo le due versioni avanzate dell'algoritmo di Eulero, ovvero la versione Bird-Salvo e, Eulero-Hamming. Gli algoritmi si occupano di eliminare dallo stream principale, rappresentante $\mathbb{N}_{\geq 2 \wedge \leq n}$, tutti i composti generati a partire dai primi identificati. L'ottimizzazione che introdurremo in questa parte, riguarda lo stream principale: utilizzando una ruota w_4 per generare lo stream iniziale, riduciamo la densità dell'insieme di numeri da controllare di circa il 77%, se comparato a $\mathbb{N}_{\geq 2 \wedge \leq n}$.

La comodità nell'utilizzo delle ruote, sta nel fatto che la loro integrazione non necessita, in genere, modifiche sostanziose all'implementazione degli algoritmi.

```
s4 = spin (circ w4) 11
ts4 = tail s4
eSH4 = 2:3:5:7:11:ts4 `sMinus` comp (drop 4 eSH4)
eSI4 = 2:3:5:7:11:ts4 `sMinus` comp (drop 4 eSI4) w4
```

Listing 2.15: Implementazioni Haskell degli algoritmi di Eulero-Bird ed Eulero-Hamming equipaggiati con ruote w_4

Attraverso le funzioni *comp* mostrate in ciascuna implementazione, abbiamo ridefinito le sequenze infinite di primi, integrando le ruote. Infine, abbiamo applicato una ruota w_4 per la generazione della lista iniziale L da filtrare alle due implementazioni fornite dal professor Salvo in [SP18].

```
s4 = spin (circ w4) 11
primes = 2:3:5:7:(sieve s4) where
  sieve (c:cs) = c:sieve' cs ss (insert (c*c) (tail es)
   empty) where
      es = map (c*) (c:cs)
      ss = cs `sMinus` es
      sieve' cs@(c:tcs) ss tbl
        | c < n = c : sieve' tcs ss' tbl'
        | otherwise = sieve' tcs ss tbl'' where
          (n, m:ms) = findMin tbl
          es = map (c*) ss
          ss' = tail (ss `sMinus` es)
          tbl' = insert (c*c) (tail es) tbl
          tbl'' = insert m ms (deleteMin tbl)
```

Listing 2.16: Implementazione Haskell dell'algoritmo di ONeills-Eulero con coda di priorità e $\mathbb{N}_{w4}$

```
s4 = spin (circ w4) 11
primes = 2:3:5:7:(sieve s4 w4) where
  sieve (c:cs) w = c:sieve' cs (nextWheel w c)
    (insert (c*c) (circ (map (c*) w)) empty) where
      sieve' cs@(c:tcs) w tbl
        | c < n = c : sieve' tcs w' tbl'
        | otherwise = sieve' tcs w tbl'' where
          (n, m:ms) = findMin tbl
          w' = nextWheel w c
          tbl' = insert (c*c) (circ (map (c*) w)) tbl
          tbl'' = insert (n+m) ms (deleteMin tbl)
```

Listing 2.17: Implementazione Haskell dell'algoritmo di ONeills-Eulero con coda di priorità, $\mathbb{N}_{w4}$ e $nextWheel$

Da Bird-Eratostene ad una modellazione efficiente dell'algoritmo di Eulero

Abbiamo visto, fino ad ora, come utilizzare le ruote per generare a priori la lista iniziale $L = [2..n]$ filtrata, oppure, come nel caso degli algoritmi di ONeills-Eulero, come generare lo stream di multipli disgiunti per ciascun primo p_k. A partire dall'implementazione dell'algoritmo di Bird-Eratostene, presentiamo ora la nostra proposta per una modellazione efficiente dell'algoritmo di Eulero. Il primo problema da affrontare riguarda la generazione dei numeri composti: è possibile generare, in modo efficiente, attraverso la struttura a liste di multipli ideata ed utilizzata da Richard Bird nella sua implementazione dell'algoritmo di Eratostene, gli stream disgiunti di multipli producendo quindi esattamente $n - \pi(n)$ numeri composti? Attraverso una reinterpretazione ricorsiva della procedura *composites*, e l'uso di *nextWheel*, abbiamo derivato una procedura che, a partire dalla lista di primi *primes*, soddisfa le due proprietà introdotte poc'anzi.

```
composites = foldr unionP [] (multiples primes [1]) where
  multiples (p:xs) w = map (p*) s : multiples xs nw where
    s  = spin (circ w) p
    nw = nextWheel w p
```

Listing 2.18: Calcolo efficiente in $O(n)$ dei numeri composti in Haskell

A partire dalla ruota $w_0 = [1]$, *multiples* genera ricorsivamente lo stream successivo di multipli appartenenti al primo p_k con elementi coprimi ai $k - 1$ primi già identificati. Ad ogni passo, la ruota successiva è generata efficientemente dalla funzione *nextWheel*. La funzione $foldr$, attraverso la procedura di unione ottimizzata *unionP*, si occupa di attraversare la lista di liste di multipli disgiunte, risultante da *multiples*, fornendo infine, la lista ordinata di numeri composti da scartare da L. Possiamo però fare di meglio: generando la lista iniziale L attraverso la ruota w_k, scartiamo a priori i multipli dei k primi iniziali. Di conseguenza, i k stream iniziali generati da *multiples*, risultano superflui. Da qui deriviamo la nostra versione altamente ottimizzata dell'algoritmo di Eulero che sfrutta la ruota w_4 e *nextWheel*.

```
primes = 2:3:5:7:11:13:(drop 2 ss) `sMinus` comp
  unionP (x:xs) ys = x:dUnion xs ys
  comp = foldr unionP [] (mult (drop 5 primes) w4)
  mult (p:xs) w = (map (p*) s) : mult xs nw where
    s  = spin (circ w) p
    nw = nextWheel w p
  w4 = genWheel [2,3,5,7]
  ss = spin (circ w4) 11
```

Listing 2.19: Implementazione Haskell dell'algoritmo di Eulero Bird-Picarella con $\mathbb{N}_{w4}$ e *nextWheel*

Questa serie di ottimizzazioni consente inoltre, di sostituire *union* e *minus* con le corrispettive procedure ottimizzate *dUnion* e *sMinus*, riducendo il numero di operazioni di confronto per ciascun elemento.

Capitolo 3

Benchmark prestazionali di generatori di primi in Haskell

Testare le performance computazionali e di memoria di un'implementazione, risulta cruciale nel caso di linguaggi funzionali come Haskell: come abbiamo introdotto nel primo capitolo, il paradigma lazy, introduce una difficoltà aggiuntiva al calcolo della reale complessità computazionale e, spesso, è necessario ricorrere ad approssimazioni. Haskell è un linguaggio ad alto livello che, da un ecosistema puramente funzionale, deve essere necessariamente mappato ad uno imperativo durante la fase di compilazione. Questo introduce, ad esempio, nel caso dell'annidamento di complementazioni di stream nell'algoritmo di Eulero naive, una serie di operazioni aggiuntive necessarie alla gestione delle strutture dati utilizzate. Pertanto, siamo fortemente interessati ad eseguire gli algoritmi proposti su istanze di test reali, definendo, alla fine di questo capitolo, l'implementazione funzionale più resistente al degrado temporale e spaziale a cui assistiamo al crescere del primo p_n richiesto.

3.1 La struttura dei test

Come nel caso dell'ordinamento, gli algoritmi funzionali sono stati incapsulati all'interno di funzioni a singolo parametro. Siamo interessati al calcolo della sequenza S_n: per forzare gli algoritmi a computare l'intera sequenza, richiediamo dalla lista *primes* generata, di estrapolare attraverso l'operatore $IndexOf$!!, il primo n-esimo, dove n, è il parametro di tipo $PositiveInt$ fornito.

```
nomeAlgoritmo :: PositiveInt -> Int
nomeAlgoritmo (PositiveInt n) = primes !! (n-1) where
  primes = #algoritmo
```

Listing 3.1: Wrapper per i test sui generatori di primi in Haskell

Come abbiamo già introdotto, ogni funzione wrapper accetta un parametro di tipo $PositiveInt$ definito come segue: dato un Int n in input, se n è negativo restituisce un errore, altrimenti, restituisce n. Definiamo, attraverso la keyword *newtype*, il nuovo tipo $PositiveInt$ e la sua istanza di $Arbitrary$. Sfruttiamo, nella definizione del metodo generatore *arbitrary*, la funzione *sized* che, come abbiamo introdotto nel

primo capitolo, permette di definire un comportamento con cui variare la cardinalità dell'input da generare attraverso il parametro intero n. Definiamo inoltre l'istanza $NFData$ in cui forniamo le istruzione con cui ridurre il tipo $PositiveInt$ in forma normale.

```
newtype PositiveInt = PositiveInt Int deriving Show

instance Arbitrary PositiveInt where
  arbitrary = sized(\n ->
    if n < 0
    then error $ "Valore negativo" ++ show n
    else return $ PositiveInt n)

instance NFData PositiveInt where
  rnf (PositiveInt n) = rnf n
```

Listing 3.2: Definizione ed istanza Arbitrary e NFData del tipo PositiveInt in Haskell

Verrà testato separatamente ciascun gruppo di algoritmi attraverso 20 istanze di input crescenti nel range $0 \leq n \leq 10^7]$. Ogni test è vincolato da un limite di 2 minuti di esecuzione e 10GB di allocazione massima simultanea nello heap. Se un test dovesse superare tali limiti, a traccia del suo fallimento, il risultato verrà riportato attraverso la dicitura *time* o *mem*. Proprio per questo, almeno per i grafici, limiteremo il range visibile nelle immagini a valori tali che, ogni algoritmo di quel determinato gruppo, sia in grado di terminare rispettando i vincoli. Per ogni gruppo riporteremo due grafici: nel primo riporteremo i tempi di esecuzione, mentre, nel secondo, le allocazioni massime di memoria. In aggiunta, forniremo delle tabelle i risultati per valori di n esterni al range d'interesse dei grafici. Il sistema operativo utilizzato durante la fase di testing è Ubuntu 20.04.1 LTS mentre la macchina è equipaggiata con 10GB di ram ed un Intel core i7 9900k.

3.1.1 Gli algoritmi elementari

Tra gli algoritmi elementari presentati, le divisioni successive, si confermano la miglior procedura funzionale per il calcolo di sequenze di numeri primi. La memoria massima allocata ed i tempi di esecuzione sono i migliori in entrambi i grafici. Mentre Turner ed Eulero naive violano rispettivamente il vincolo di tempo e spazio imposto, andando oltre $2 \cdot 10^4$, l'algoritmo delle divisioni successive si dimostra resistente e, solo per $n = 1.4 \cdot 10^5$, il degrado prestazionale lo rende impraticabile in termini di tempo. Come abbiamo introdotto nel secondo capitolo, dato un numero c, l'algoritmo delle divisioni successive esegue un numero di controlli nell'ordine di $O(\pi(\sqrt{c}))$. D'altra parte, il numero di controlli eseguiti da Turner è lineare rispetto al primo richiesto, degradando le prestazioni molto più velocemente. L'algoritmo di Eulero naive detiene l'andamento peggiore in termini di tempo e spazio. Mentre una procedura imperativa $minusa, b$ richiede $O(|a|)$ operazioni, nel caso funzionale, scorrendo entrambe le liste, sono necessari $O(|a \cup b|)$ passi, evidenziando quindi una prima inefficienza sui tempi. Ad ogni nuovo primo p_k scoperto, l'implementazione di Eulero naive avvia una nuova operazione di complementazione attraverso la procedura $minus$, causando ulteriori allocazioni e deallocazioni, dovute al percorso che ogni numero c, a partire dalla

lista iniziale L, deve compiere fino a raggiungere la testa dell'ultima lista filtrata nel caso c sia primo; altrimenti, al massimo, lo stream complementato appartenente al primo $p_{\pi(\sqrt{c})}$, nel caso c fosse un composto. Sebbene questo avvenga secondo una logica lazy in cui la memoria viene immediatamente deallocata se non necessaria, la serie di allocazioni e deallocazioni necessarie al funzionamento dell'algoritmo necessitano di ulteriore memoria dedicata al funzionamento del Garbage Collector e, questa serie di operazioni, al crescere del primo p_k richiesto, richiedono sempre più tempo. Notiamo infine un particolare fondamentale: Eulero naive genera una serie di complementazioni di stream annidati. In questo annidamento, lo stream derivante dal primo p_k è dipendente dai $k-1$ stream precedenti. La gestione di questa struttura è affidata ad Haskell attraverso l'uso di strutture dati dedicate. Sebbene non sia semplice definire come Haskell gestisca tale struttura e sopratutto che tipologia di struttura sia, attraverso i report di tempo e spazio, siamo certi che la sua manipolazione introduca una quantità consistente di lavoro aggiuntivo che porta ben presto, a partire da $n > 2{\cdot}10^4$, l'implementazione ad essere inefficiente.

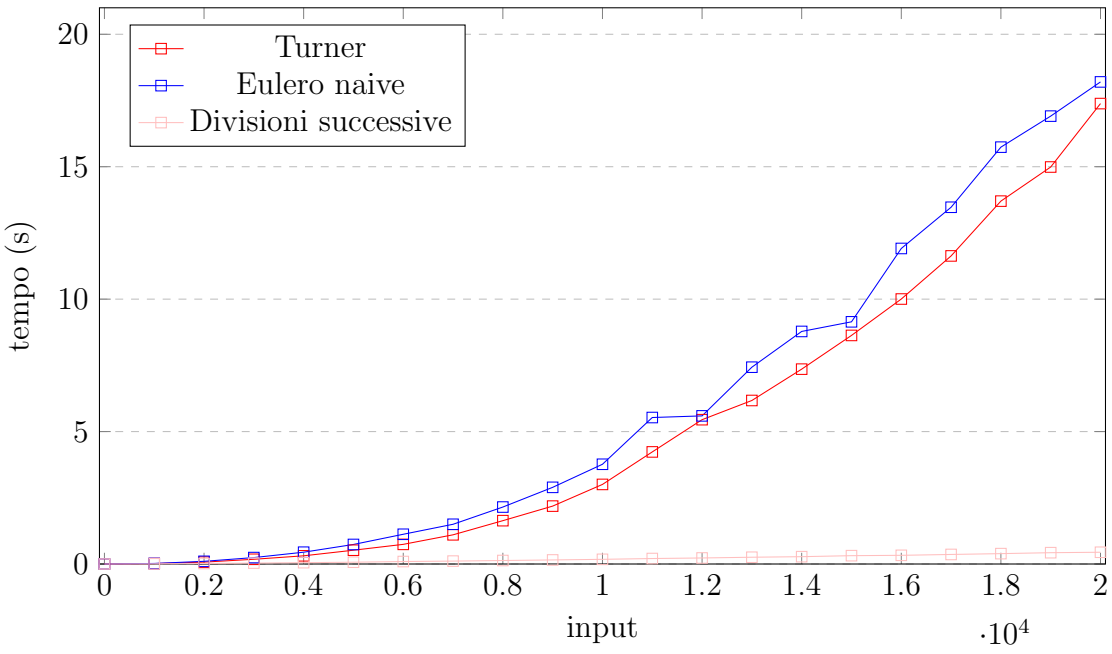
Algoritmo	$2 \cdot 10^4$	$2 \cdot 10^5$	$1.4 \cdot 10^6$	$4 \cdot 10^6$	$6 \cdot 10^6$	10^7
Turner	17.383	time	-	-	-	-
Eulero Naive	18.201	mem	-	-	-	-
Divisioni successive	0.443	9.570	time	-	-	-

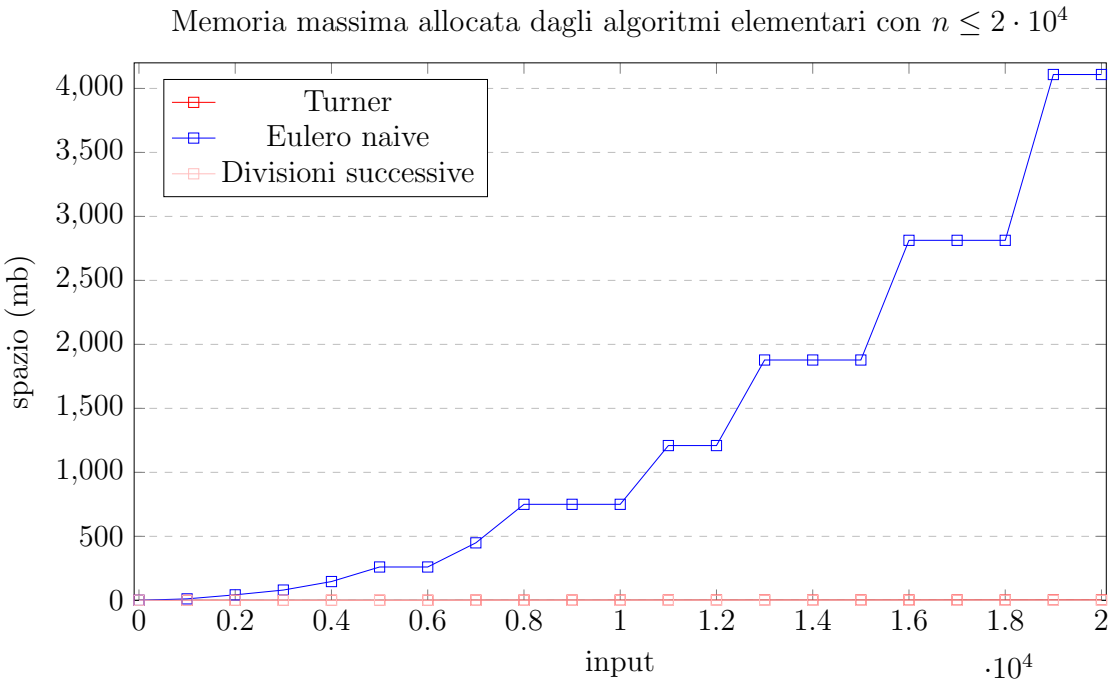
Tabella 3.1: Tempo di esecuzione degli algoritmi elementari con $2 \cdot 10^4 \leq n \leq 10^7$

Algoritmo	$2 \cdot 10^4$	$2 \cdot 10^5$	$1.4 \cdot 10^6$	$4 \cdot 10^6$	$6 \cdot 10^6$	10^7
Turner	2.375	time	-	-	-	-
Eulero Naive	4108.829	mem	-	-	-	-
Divisioni successive	0.069	6.455	time	-	-	-

Tabella 3.2: Memoria massima allocata dagli algoritmi elementari con $2{\cdot}10^4 \leq n \leq 10^7$

Tempo di esecuzione degli algoritmi elementari con $n \leq 2 \cdot 10^4$

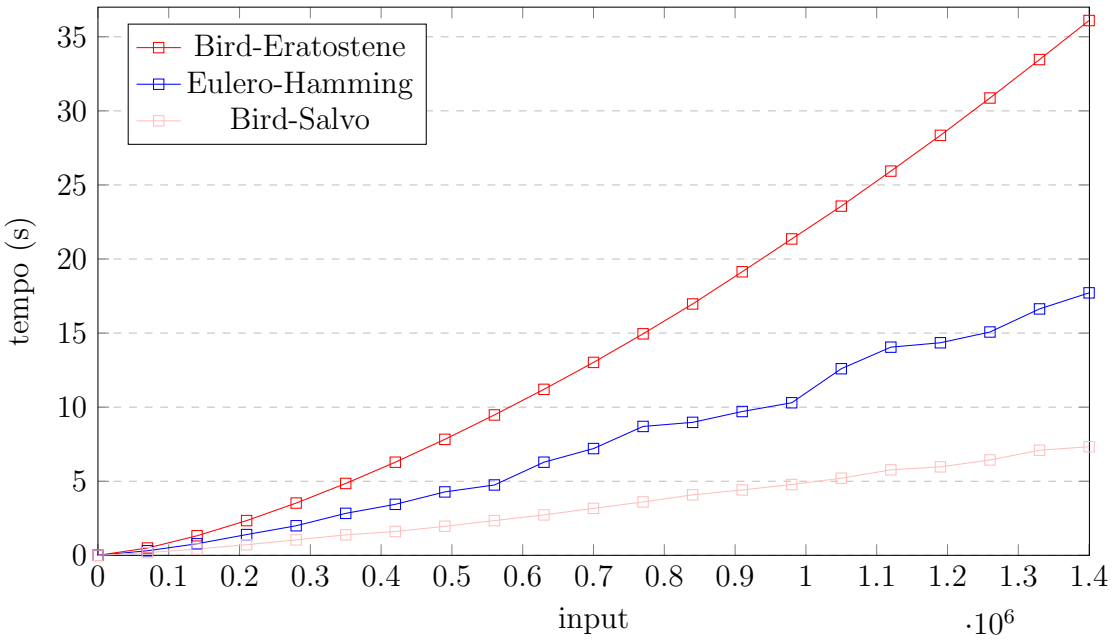




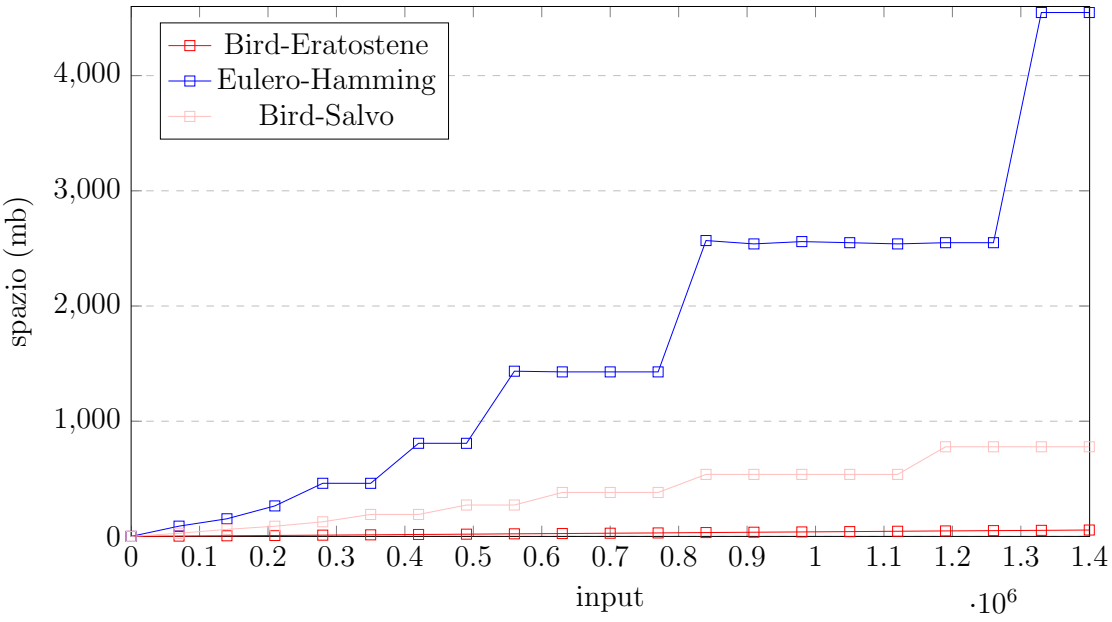
3.1.2 Gli algoritmi avanzati

L'algoritmo di Bird-Eratostene riporta il peggior andamento della curva temporale tra i 3 algoritmi avanzati. Come avevamo previsto nel secondo capitolo, il limite principale riguarda la procedura di *union* nella fase in cui i k stream di multipli devono essere uniti in un unico stream ordinato. La curva riguardante lo spazio massimo allocato riporta però degli ottimi risultati: ad ogni passo, avendo identificato k primi, l'algoritmo necessita solo dei k valori in testa a ciascun stream di multipli più il primo numero delle sequenze L (iniziale) e dei composti ordinati. Nonostante la complessità computazionale teorica sia peggiore rispetto all'algoritmo delle divisioni successive, l'algoritmo di Bird-Eratostene si è dimostrato più efficiente in tutto il range ricoperto dal suddetto e maggiormente scalabile, arrivando fino ad $n = 1.4 \cdot 10^6$. L'algoritmo di Eulero-Hamming ottiene invece dei tempi 2x migliori rispetto a Bird-Eratostene. La prima ottimizzazione introdotta riguarda la funzione di *union*: sfruttando la proprietà che *head* $s_k <$ *head* s_{k+1} con s_k, s_{k+1} gli stream adiacenti ad una *union*, l'algoritmo esclude il caso $x > y$, riducendo il lavoro svolto per ogni singolo elemento. Tuttavia, l'algoritmo impiega un numero di *union* lineare rispetto all'indice k del primo p_k richiesto. Come nel caso dell'algoritmo di Eulero naive, assistiamo, nel grafico dedicato allo spazio allocato, allo stesso pattern incrementale. Anche qui, la gestione delle operazioni di stream fusion introduce un overhead nella memoria allocata, dovuto al lavoro svolto dal garbage collector di ghc. Ciononostante, gli overhead sui tempi, in questo caso, sono contenuti e consentono all'algoritmo di raggiungere il primo $p_{1.4 \cdot 10^6}$. Nel caso dell'algoritmo di Bird-Salvo, la reinterpretazione dell'equazione alla base dell'algoritmo di Eulero naive, ha permesso di ottenere degli ottimi tempi e delle allocazioni in memoria piuttosto contenute. In questo caso infatti, abbiamo eliminato l'annidamento di complementazioni tra stream ed ottimizzato la procedura *minus* sfruttando la proprietà *head* $L <$ *head composites*.

Tempo di esecuzione degli algoritmi avanzati con $n \leq 1.4 \cdot 10^6$



Memoria massima allocata dagli algoritmi avanzati con $n \leq 1.4 \cdot 10^6$



Algoritmo	$2 \cdot 10^4$	$2 \cdot 10^5$	$1.4 \cdot 10^6$	$4 \cdot 10^6$	$6 \cdot 10^6$	10^7
Bird-Eratostene	0.093	2.136	35.594	time	-	-
Eulero-Hamming	0.083	1.579	17.939	mem	-	-
Eulero Bird-Salvo	0.050	0.706	7.503	30.783	56.065	81.310

Tabella 3.3: Tempo di esecuzione degli algoritmi avanzati con $2 \cdot 10^4 \leq n \leq 10^7$

Algoritmo	$2 \cdot 10^4$	$2 \cdot 10^5$	$1.4 \cdot 10^6$	$4 \cdot 10^6$	$6 \cdot 10^6$	10^7
Bird-Eratostene	0.713	7.791	55.542	time	-	-
Eulero-Hamming	18.329	264.823	4547.675	mem	-	-
Eulero Bird-Salvo	8.574	88.464	778.158	2487.695	3940.986	mem

Tabella 3.4: Memoria massima allocata dagli algoritmi avanzati con $2 \cdot 10^4 \leq n \leq 10^7$

3.1.3 Gli algoritmi ottimizzati

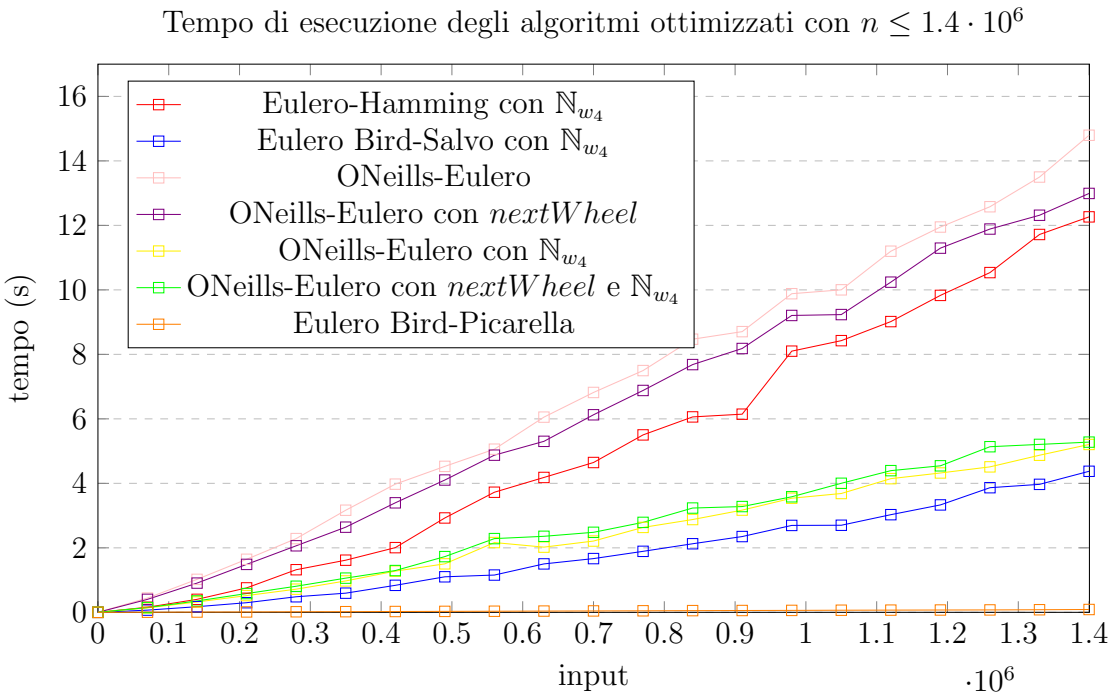
L'implementazione fornita in Eulero Bird-Picarella, a partire da quella di Bird-Eratostene, si dimostra, per questo benchmark, la migliore in termini di tempo e spazio massimo allocato simultaneamente. La densità della lista iniziale $L = [2..n]$, contrariamente alla sua versione imperativa, è ridotta di circa il 77%. Inoltre, si osserva che, l'esclusione dei primi 4 stream di multipli nel calcolo dei composti, fa conseguire un ulteriore miglioramento delle performance. Non ultimo fattore di velocizzazione, consegue dall'utilizzo delle versioni ottimizzate di $dUnion$ e $sMinus$, le quali, riducono il numero di operazioni eseguite per ciascun composto generato. L'algoritmo di Bird-Salvo, equipaggiato con la ruota w_4, che gli consente di risparmiare molto del lavoro svolto rispetto al suo predecessore non ottimizzato, segue in ordine di performance in termini di tempo e spazio massimo allocato simultaneamente. A seguire, i due algoritmi di ONeills-Eulero ottimizzati con w_4 e $nextWheel$ mantengono un andamento piuttosto simile, sebbene, la versione che non sfrutta $nextWheel$, detiene dei tempi leggermente inferiori dovuti al lavoro risparmiato nella gestione e calcolo delle ruote. L'algoritmo di Eulero-Hamming invece, nonostante qualche miglioramento nei tempi di esecuzione, è limitato dall'elevata quantità di memoria allocata, anche nella sua versione con ruota w_4, superando, a partire dal test per $n = 4 \cdot 10^6$, il vincolo imposto riguardante la memoria massima allocata simultaneamente.

Algoritmo	$2 \cdot 10^4$	$2 \cdot 10^5$	$1.4 \cdot 10^6$
Eulero-Hamming con $\mathbb{N}_{w4}$	0.035	0.807	12.264
Eulero Bird-Picarella	0.001	0.011	0.090
Eulero Bird-Salvo con $\mathbb{N}_{w4}$	0.022	0.315	4.374
ONeill-Eulero	0.121	1.605	14.795
ONeill-Eulero con $nextWheel$	0.095	1.348	12.991
ONeill-Eulero con $\mathbb{N}_{w4}$	0.041	0.545	13.050
ONeill-Eulero con $nextWheel$ e $\mathbb{N}_{w4}$	0.039	0.522	5.385

Tabella 3.5: Tempo di esecuzione degli algoritmi ottimizzati con $2 \cdot 10^4 \leq n \leq 1.4 \cdot 10^6$

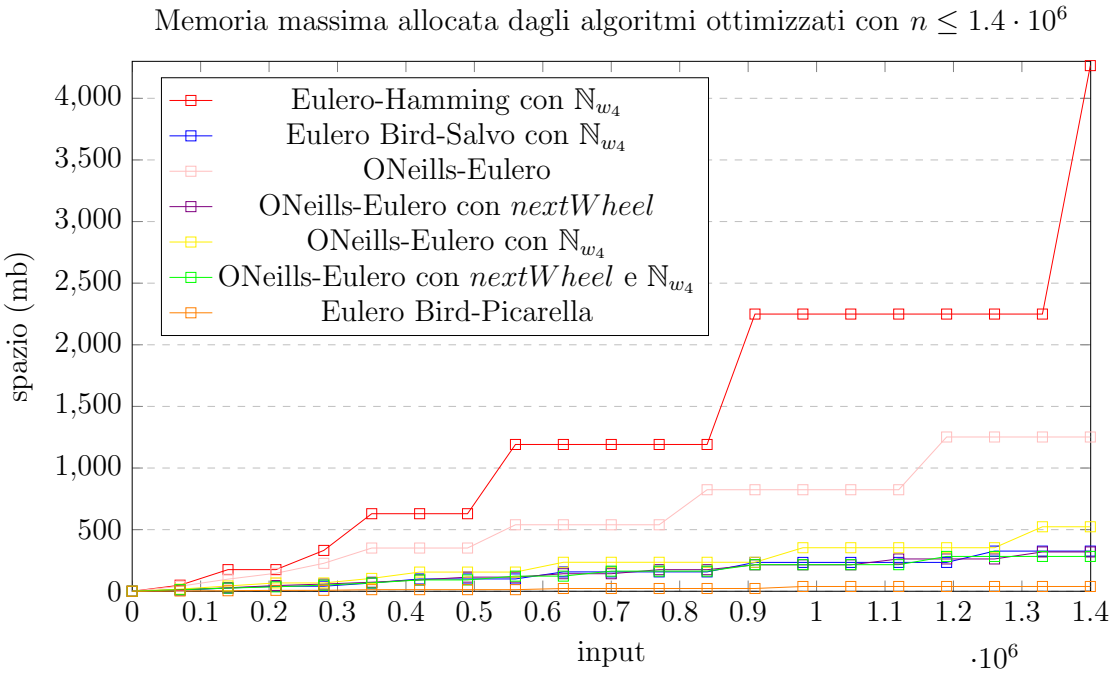
Algoritmo	$4 \cdot 10^6$	$6 \cdot 10^6$	10^7
Eulero-Hamming con $\mathbb{N}_{w4}$	mem	-	-
Eulero Bird-Picarella	0.269	0.359	0.579
Eulero Bird-Salvo con $\mathbb{N}_{w4}$	20.605	38.077	mem
ONeill-Eulero	50.104	mem	-
ONeill-Eulero con $nextWheel$	43.089	69.076	124.010
ONeill-Eulero con $\mathbb{N}_{w4}$	17.757	28.391	52.340
ONeill-Eulero con $nextWheel$ e $\mathbb{N}_{w4}$	18.615	30.382	59.302

Tabella 3.6: Tempo di esecuzione degli algoritmi ottimizzati con $4 \cdot 10^6 \leq n \leq 10^7$



Algoritmo	$2 \cdot 10^4$	$2 \cdot 10^5$	$1.4 \cdot 10^6$
Eulero-Hamming con $\mathbb{N}_{w4}$	7.139	175.522	4265.293
Eulero Bird-Picarella	0.499	7.508	38.802
Eulero Bird-Salvo con $\mathbb{N}_{w4}$	3.651	40.899	325.978
ONeill-Eulero	16.351	146.123	1251.864
ONeill-Eulero con *nextWheel*	4.013	46.271	319.069
ONeill-Eulero con $\mathbb{N}_{w4}$	5.828	67.615	523.411
ONeill-Eulero con *nextWheel* e $\mathbb{N}_{w4}$	3.883	35.984	282.678

Tabella 3.7: Memoria massima allocata dagli algoritmi ottimizzati con $2 \cdot 10^4 \leq n \leq 1.4 \cdot 10^6$



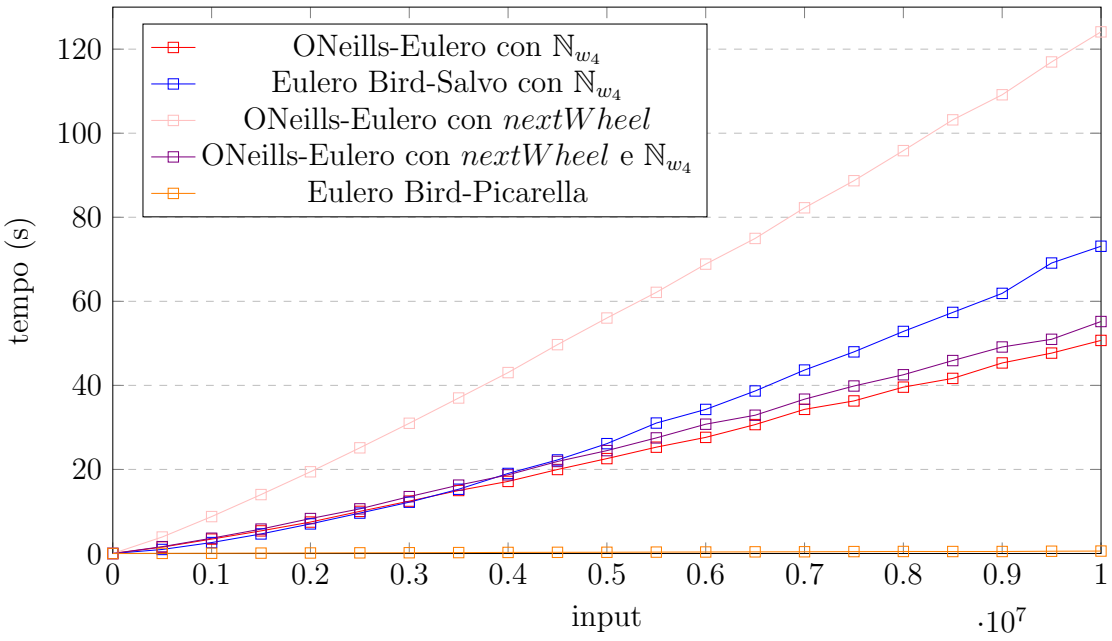
Algoritmo	$4 \cdot 10^6$	$6 \cdot 10^6$	10^7
Eulero-Hamming con $\mathbb{N}_{w4}$	mem	-	-
Eulero Bird-Picarella	114.267	195.600	334.631
Eulero Bird-Salvo con $\mathbb{N}_{w4}$	962.767	1461.306	2773.902
ONeill-Eulero	4024.225	mem	-
ONeill-Eulero con $nextWheel$	956.227	1347.343	2594.204
ONeill-Eulero con $\mathbb{N}_{w4}$	1603.729	2292.379	3249.408
ONeill-Eulero con $nextWheel$ e $\mathbb{N}_{w4}$	992.406	1579.666	2459.051

Tabella 3.8: Memoria massima allocata dagli algoritmi ottimizzati con $4 \cdot 10^6 \leq n \leq 10^7$

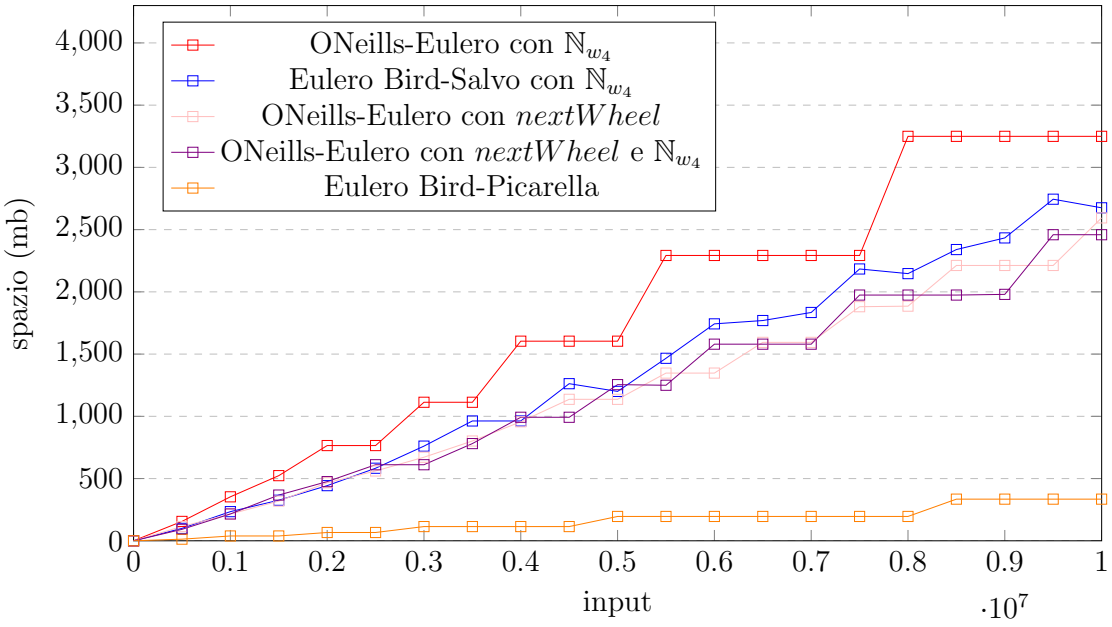
3.2 Considerazioni finali sugli algoritmi

In generale, attraverso i benchmark eseguiti fino ad ora, abbiamo assistito ad un deterioramento generale delle prestazioni, provocato maggiormente dall'incremento dell'attività di garbage collection, la quale provoca un calo della produttività. L'utilizzo delle ruote, ha solamente rimandato la problematicità riscontrata, la quale tuttavia si ripresenta già a partire da istanze di n relativamente ridotte. L'analisi, insieme ai test condotti, sull'implementazione di Bird-Picarella, palesa in modo inequivocabile, la sua elevata efficienza e scalabilità: se confrontato con le migliori performance degli altri algoritmi presi in considerazione, nel caso $n = 10^7$, il guadagno ottenuto è pari a 100x per il tempo di esecuzione e, 7x per lo spazio massimo allocato simultaneamente. Di conseguenza, l'implementazione dell'algoritmo di Eulero di Bird-Picarella si è selettivamente mostrata la proposta funzionale più efficiente atta alla generazione di sequenze di numeri primi in Haskell.

Tempo di esecuzione degli algoritmi resistenti per $n \leq 10^7$

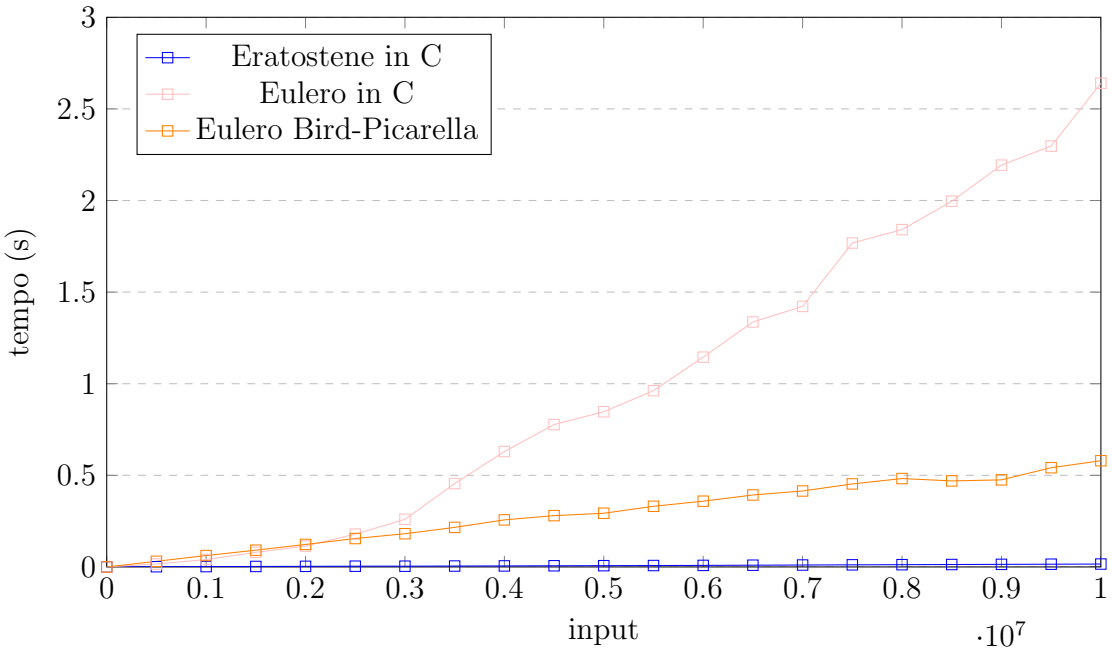


Memoria massima allocata dagli algoritmi resistenti per $n \leq 10^7$

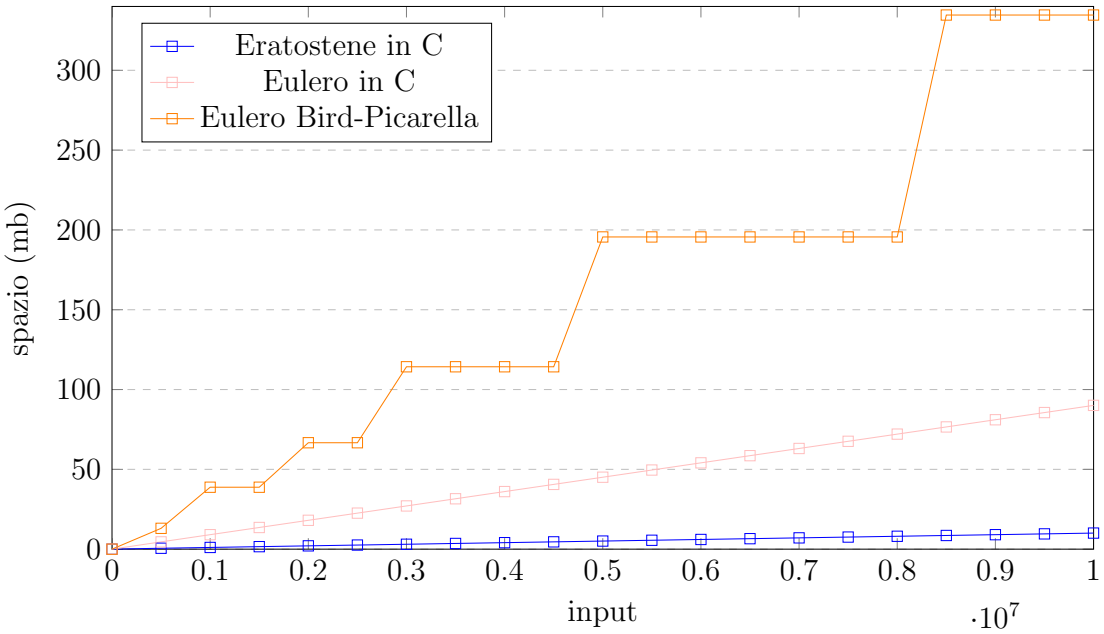


Infine, proponiamo un ultimo benchmark che mette a confronto le due implementazioni imperative in C introdotte nel secondo capitolo, riguardanti l'algoritmo di Eratostene ed Eulero, con l'implementazione eletta di Bird-Picarella. I risultati sono eccellenti: la versione funzionale supera di gran lunga i tempi ottenuti da Eulero in C. Sebbene la versione imperativa dell'algoritmo di Eratostene ottenga dei tempi e spazio decisamente migliori rispetto alla versione funzionale, possiamo considerare l'implementazione funzionale eletta poc'anzi globalmente efficiente. Questo dimostra che l'utilizzo sinergico del linguaggio funzionale Haskell e l'efficienza del compilatore GHC possano consentire l'implementazione di algoritmi funzionali, ad alto livello, ed efficienti.

Tempo di esecuzione delle implementazioni C ed Eulero Bird-Picarella



Memoria massima allocata dalle implementazioni C ed Eulero Bird-Picarella



Algoritmo	$2 \cdot 10^4$	$2 \cdot 10^5$	$1.4 \cdot 10^6$	$4 \cdot 10^6$	$6 \cdot 10^6$	10^7
Eratostene in C	0	0.001	0.002	0.005	0.008	0.015
Eulero in C	0.002	0.005	0.068	0.630	1.145	2.640
Eulero Bird-Picarella	0.001	0.011	0.090	0.269	0.359	0.579

Tabella 3.9: Tempo di esecuzione delle implementazioni C ed Eulero Bird-Picarella

Algoritmo	$2 \cdot 10^4$	$2 \cdot 10^5$	$1.4 \cdot 10^6$	$4 \cdot 10^6$	$6 \cdot 10^6$	10^7
Eratostene in C	0.092	0.272	1.472	0.005	0.008	0.015
Eulero in C	0.252	1.872	12.672	0.630	1.145	2.640
Eulero Bird-Picarella	0.499	7.508	38.802	114.267	195.600	334.631

Tabella 3.10: Memoria massima allocata dalle implementazioni C ed Eulero Bird-Picarella

Capitolo 4

Conclusioni finali

Nel primo capitolo abbiamo introdotto il lettore al paradigma di programmazione funzionale con riferimenti ad Haskell. Sfruttando il caso di studio dell'ordinamento, abbiamo presentato gli strumenti di benchmarking e profiling con cui analizzare le performance di programmi Haskell. Nel secondo capitolo, dopo aver definito formalmente i numeri primi, attraverso alcuni dei risultati derivati dal teorema dei numeri primi, abbiamo presentato ed analizzato gli algoritmi funzionali, scelti per il calcolo di sequenze di numeri primi. Nel terzo capitolo abbiamo infine, testato le performance degli algoritmi funzionali, attraverso una serie di benchmark. Sfruttando le possibilità offerte da Haskell, siamo stati in grado, nel caso degli algoritmi elementari, di implementare soluzioni algoritmiche complesse attraverso una sintassi compatta ed elegante. Questo vantaggio pone d'altra parte, un limite alle performance ottenibili dagli algoritmi che, già per $n > 2 \cdot 10^4$, li rendono impraticabili e, piuttosto, un emblema del paradigma di programmazione funzionale. Fornire nativamente la laziness e l'annidamento di stream, richiede l'impiego di strutture dati ed algoritmi dedicati alla loro gestione dietro le quinte. In molteplici situazioni in cui il loro utilizzo è massiccio, un algoritmo all'apparenza efficiente, come ad esempio l'algoritmo di Eulero naive, può rivelarsi inefficiente. Si è evinto che, modellare un algoritmo di sieving imperativo in un'implementazione funzionale efficiente, è un processo complesso. Nel caso dell'algoritmo di Eratostene, è stato necessario modellare la marcatura dei composti, attraverso una sottrazione tra insiemi, con un conseguente incremento della complessità computazionale originaria, dovuto all'impiego della procedura di stream fusion *union*. D'altra parte, nel caso dell'algoritmo di Eulero, partendo da una versione imperativa non banale, abbiamo ottenuto un'implementazione funzionale naive estremamente semplice e compatta. Si è inoltre dimostrato che, è possibile ottenere implementazioni funzionali efficienti per il calcolo di sequenze di numeri primi: sfruttando le ruote e code di priorità, abbiamo superato i limiti imposti dalle operazioni di fusione tra stream e ridotto la memoria allocata, fornendo delle varianti dell'algoritmo di Eulero altamente scalabili come ad esempio l'implementazione di ONeills-Eulero con $nextWheel$ e $\mathbb{N}_{w4}$ proposta in [SP18]. Alla fine del secondo capitolo, abbiamo presentato una modellazione funzionale inedita dell'algoritmo di Eulero, basata sull'implementazione di Bird-Eratostene. Attraverso una serie di benchmark su reali istanze di input, abbiamo eletto l'implementazione dell'algoritmo di Eulero Bird-Picarella la versione più efficiente e scalabile per il calcolo di sequenze di numeri primi tra gli algoritmi funzionali proposti.

Bibliografia

[Ben20] Google Benchmark. https://github.com/google/benchmark, 2020. Consulted September 2020.

[HH18] Martin Handley and Graham Hutton. Autobench: comparing the time performance of haskell programs. *ACM SIGPLAN Notices*, 53:26–37, 09 2018.

[NHGW18] Arthur Nunes-Harwitt, Matthew Gambogi, and Travis Whitaker. Quicksort: A pet peeve. pages 547–549, 02 2018.

[nHw20] Prime numbers (Haskell wiki). https://wiki.haskell.org/Haskell, 2020. Consulted October 2020.

[O'N09] Melissa O'Neill. The genuine sieve of eratosthenes. *J. Funct. Program.*, 19:95–106, 01 2009.

[Pag20a] Haskell Home Page. https://wiki.haskell.org/, 2020. Consulted October 2020.

[Pag20b] PQueue Hackage Page. https://hackage.haskell.org/package/pqueue, 2020. Consulted October 2020.

[Rib97] Paulo Ribenboim. Are there functions that generate prime numbers? *The College Mathematics Journal*, 28(5):352–359, 1997.

[SP18] Ivano Salvo and Agnese Pacifico. Three euler's sieves and a fast prime generator (functional pearl), 2018.

[Wik20] Foreign Function Interface (Haskell Wiki). http://wiki.haskell.org/Foreign_Function_Interface, 2020. Consulted October 2020.