UNIVERSITÀ DEGLI STUDI DÌ UDINE

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

RELAZIONE PROGETTO DÌ ALGORITMI E STRUTTURE DATI

A cura di:

- Raffaele Marseu, matricola 133879
- Maksym Sernyak, matricola 143087
- Danjel Stefani, matricola 142863
- Marco Venir, matricola 144762

Anno Accademico 2019/2020

Indice

1. Introduzione

Abbiamo realizzato i programmi in Java sul programma Intellij, come ambiente di sviluppo e di run dei programmi. Abbiamo usato un computer con uno Xeon W3680 (3.33 GHz di base, 3.60 GHz Intel TurboBoost 1.0) con 6 core e 12 threads, equipaggiato con 20gb di ram DDR3 1333 MHz di velocita'. Il sistema operativo usato e' Windows 10 Pro. Come JDK abbiamo usato la versione 14.0.0.

2. Presentazione algoritmi

2.1. QuickSelect

```
public static int QuickSelect(ArrayList<Integer> array, int inizio, int fine, int k)
{
    if (inizio == fine)
    {
        return array.get(inizio);
    }

    int indice = partition(array, inizio, fine);

    if (k == indice)
    {
        return array.get(k);
    }
    else if (k < indice)
    {
        return QuickSelect(array, inizio,  fine: indice - 1, k);
    }
    else
    {
        return QuickSelect(array,  inizio: indice + 1, fine, k);
    }
}
```

Nel codice di QuickSelect controlliamo se è presente un solo elemento e in caso affermativo lo ritorniamo. *"partition"* (codice omesso, perché da manuale) divide l'array in varie parti e nella variabile "indice" salva il valore dell'elemento di pivot, il quale viene confrontato con k. Se il valore di entrambe le variabili coincide, ritorna l'elemento che si trova in posizione di k.
In seguito, controlliamo se l'indice trovato con partition e' maggiore di k; se si, richiamiamo Quickselect con la variabile "fine" uguale a indice - 1. Altrimenti richiamiamo Quickselect con la variabile "inizio" uguale a indice + 1.

2.2. Heap Select

Per implementare questo algoritmo abbiamo utilizzato la classe "*nodo*" per memorizzare la posizione e il valore di ogni elemento nella heap in modo da rendere unico ogni nodo ed evitare problemi di ricerca (ad esempio in caso di più nodi con valore uguale).

```
public static class nodo
{
    public Integer key;
    private int pos;

    public nodo(int key, int pos)
    {
        this.key = key;
        this.pos = pos;
    }

    public int getKey()
    {
        return key;
    }

    public int getPos()
    {
        return pos;
    }
}
```

```
public static void heapSelect(ArrayList<nodo> h1, int k)
{
    BuildMinHeap(h1);
    ArrayList<nodo> h2 = new ArrayList<>();
    h2.add(new nodo(h1.get(0).getKey(), h1.get(0).getPos()));

    for (int i = 0; i < k - 1; i++)
    {
        int parente = h2.get(0).getPos();
        int fSinistro = Left(parente);
        int fDestro = Right(parente);

        scambia1(h2, a: 0,  b: h2.size() -1);
        h2.remove( index: h2.size()-1);
        MinHeapify1(h2,  i: 0);

        if (fSinistro < h1.size())
        {
            Inserimento(h2, h1.get(fSinistro));
        }
        if (fDestro < h1.size())
        {
            Inserimento(h2, h1.get(fDestro));
        }

    }
    System.out.println(h2.get(0).key);
}
```

"HeapSelect", all'inizio costruisce la heap "h1" a partire dall'array fornito con BuildMinHeap (codice omesso, perché da manuale). Dopodiché si crea la heap "h2" inizialmente vuota. In essa viene inserito il primo elemento della heap *"h1"*. Durante ogni ciclo *for*, si calcolano le variabili *"figlio sinistro"* e *"figlio destro"* (codici omessi, perché da manuale). In seguito, si scambiano il primo e l'ultimo nodo (con il metodo *"scambia1"*) e si rimuove l'ultimo nodo dalla heap. A questo punto, se esistono i figli, si salvano nella heap *"h2"*.

```
public static void scambia1(ArrayList<nodo> array, int a, int b)

{
    nodo aa, bb;

    aa = array.get(a);
    bb = array.get(b);

    array.set(a, bb);
    array.set(b, aa);
}
```

```java
public static void scambia(ArrayList<nodo> array, int a, int b)

{
    int a0 = array.get(a).getKey();
    int b0 = array.get(a).getPos();

    int a1 = array.get(b).getKey();
    int b1 = array.get(b).getPos();

    array.set(a, new nodo(a1, b0));
    array.set(b, new nodo(a0, b1));
}
```

La funzione *"scambia1*" scambia semplicemente i due nodi all'interno della heap *"h2"* mentre *"scambia"*, scambia le variabili *"key"* che rappresentano i valori dei nodi. Tale funzione viene utilizzata soltanto durante la costruzione della heap *"h1"*.

"MinHeapify1" è identico a "*MinHeapify"* da manuale. L'unica differenza tra i due è che "*Minheapify1"* usa "*scambia1"* nel codice.

Per caricare i figli nella heap *"h2"* viene usata la funzione "*Inserimento*".

```java
public static void Inserimento(ArrayList<nodo> array, nodo s)
{
    array.add(s);
    FaiSalire(array, posizione: array.size()-1);
}
```

Il metodo "*Inserimento*" richiama la funzione "*FaiSalire*". Questa funzione, controlla se il valore dell'elemento è minore del valore del genitore e in tal caso scambia i due nodi. Ripete ricorsivamente *"FaiSalire"* dalla posizione del genitore fino a quando il valore non si trova nella posizione giusta, ovvero è più grande del proprio genitore oppure è nella prima posizione.

```java
public static void FaiSalire(ArrayList<nodo> array, int posizione){
    int PosParente = Parente(posizione);
    int ValParente = array.get(PosParente).getKey();

    if(posizione == 0){
        return;
    }

    if(array.get(posizione).getKey() <= ValParente){
        scambia1(array, PosParente, posizione);
        FaiSalire(array, PosParente);
    }
}
```

2.3. Median of Medians Select

Questo algoritmo sfrutta la mediana delle mediane per trovare l'elemento richiesto.

```
public static int MedianSelect(ArrayList<Integer> array, int inizio, int fine, int k) {

    if (k > 0 && k <= fine - inizio + 1) {

        int a = fine - inizio + 1;
        ArrayList<Integer> arraymediane = new ArrayList<>();
        int i;
        for (i = 0; i < a / 5; i++) {

            arraymediane.add(Mediana(array, inizio: inizio + i * 5, fine: 5));

        }

        if (i * 5 < a) {

            arraymediane.add(Mediana(array, inizio: inizio + i * 5, fine: a % 5));

        }

        int mdm;

        if (i == 1) {

            mdm = arraymediane.get(i - 1);

        } else {

            mdm = MedianSelect(arraymediane, inizio: 0, fine: i - 1, k: i / 2);

        }

        int posizione = Partizione(array, inizio, fine, mdm);

        if (posizione - inizio == k - 1) {

            return array.get(posizione);

        }

        if (posizione - inizio > k - 1) {

            return MedianSelect(array, inizio, fine: posizione - 1, k);

        }

        return MedianSelect(array, inizio: posizione + 1, fine, k: k - posizione + inizio - 1);

    }

    return -1;
```

In primo luogo, confronta il valore di k con la dimensione dell'array che contiene le mediane. Nella condizione usiamo "fine - inizio + 1" in quanto indica il numero di elementi nell'array che parte da zero. Il ciclo *for* calcola le posizioni facendo una suddivisione nell'array originale in gruppi da 5 elementi. La suddivisione dell'array potrebbe non essere perfetta in quanto potrebbe non contenere 5 elementi e la mediana di questi viene calcolata con un if a parte. Dopodiché inizializziamo la variabile *"mdm"* che serve per memorizzare il valore della mediana delle mediane. La variabile *"mdm"* è inizializzata separatamente, come la variabile *"i"*, in quanto le utilizzeremo in seguito.
La variabile posizione viene calcolata dalla funzione "*Partizione*" (partition modificata). Se posizione - inizio corrisponde a k-1, viene ritornato il valore che si trova nella posizione *"posizione"* dell'array. Se invece è maggiore, facciamo una ricorsione della funzione con la variabile *"fine"* modificata con il valore "posizione - 1". Nel caso nessuna delle due condizioni venga soddisfatta, la funzione esegue una ricorsione con la variabile "inizio" modificata con il valore "posizione + 1" e la variabile *"k"* modificata con il valore "k - posizione + inizio - 1".

```
public static int Partizione(Integer[] array, int inizio, int fine, int mdm)
{
    for (int i = inizio; i < fine; i++)
    {
        if (array[i] == mdm)
        {
            scambia(array, i, fine);
            break;
        }
    }
    int pivot = array[fine];
    int contatore = inizio;
    for (int j = inizio; j <= fine; j++)
    {
        if (array[j] < pivot)
        {
            scambia(array, contatore, j);
            contatore = contatore + 1;
        }
    }
    scambia(array, fine, contatore);
    return contatore;
}
```

Nella funzione *"Partizione*" il for iniziale sposta la variabile *"mdm"* in posizione *"fine"*. Il pivot corrisponde alla nostra variabile *"mdm"* e viene eseguita una normale funzione di partizione (identica a quella di QuickSelect). Il codice della funzione "*scambia*" è omesso, data la semplicità del codice in quanto scambia gli elementi, date le posizioni nell'array.

3. La funzione "*Creatore*"

In questa funzione utilizziamo il "*Random*" predisposto dalla libreria di Java. Impostiamo i limiti inferiore e superiore che rappresentano il minimo e massimo numero che il *"Random"* può selezionare. Per fare in modo che possano essere presenti anche numeri negativi, utilizziamo la variabile *"a*" in modo che l'intervallo di scelta sia *"y"*. Nella variabile *"y"*, il "+1" viene aggiunto, perché la funzione "nextInt()" non considera l'ultimo elemento del range. Sottraendo il limite inferiore, sarà possibile avere anche numeri negativi nell'array.

```java
public static ArrayList creatore(int n)
{
    Random random = new Random();
    int limiteI = -100000000;
    int limiteS = 100000000;
    int cicli = n;
    int y = limiteS - limiteI + 1;

    ArrayList<Integer> array = new ArrayList<>();

    for (int i = 0; i < cicli; i++)
    {
        int a = random.nextInt(y) + limiteI;
        array.add(a);
    }
    return array;
}
```

Ad esempio:
Limite Inferiore = -100;
Limite superiore = 100;
Range Y = Limite Superiore - Limite Inferiore + 1 = 100 + (-100) + 1 = 201;
Random Y = 10;
Numero finale = Random Y + Limite inferiore = 10 + (-100) = -90;

Nell'implementazione di tutti gli algoritmi, abbiamo utilizzato un range che va da -100.000.000 a 100.000.000.

4. Il metodo per misurare il tempo

Per granularità (o risoluzione) si intende l'intervallo di misurazione necessario al sistema per caricare una singola istruzione, in questo caso noi carichiamo e tentiamo di ottenere il tempo minimo registrabile e ritorniamo la differenza tra i due risultati.

```java
public static long Granularità() {

    long t0 = System.nanoTime();
    long t1 = System.nanoTime();

    while (t1 == t0){

        t1 = System.nanoTime();

    }
    return (t1 - t0);
}
```

```java
public static double ripCalculateTare(int d, long tMin){

    long t0 = 0;
    long t1 = 0;
    long rip = 1;

    while (t1 - t0 <= tMin){
        rip = rip * 2;
        t0 = System.currentTimeMillis();
        for (int i = 1; i <= rip; i++){
            prepare(d);
        }
        t1 = System.currentTimeMillis();
    }

    long max=rip;
    long min=rip/2;
    int failedCycles=5;
    while((max-min)>=failedCycles) {
        rip= (max+min)/2; //medium value
        t0=System.currentTimeMillis() ;
        for(int i=0; i<=rip; i++) {
            prepare(d);
        }
        t1=System.currentTimeMillis();
        if(t1-t0<= tMin) {
            min=rip;
        }else {
            max=rip;
        }
    }
    return max;
}
```

In "*ripCalculateTare*" calcoliamo il numero di ripetizioni minimo per far si che l'esecuzione di *"prepare"* sia registrato come tempo di preparazione e venga registrato come dato per conoscere il tempo per il caricamento dati tenendo conto anche dei cicli errati. Il tempo di *"prepare"* viene registrato per poterlo sottrarre al tempo totale ed avere il tempo di esecuzione vero e proprio dell'algoritmo.
Nel primo while verifichiamo
ogni quante ripetizioni la differenza dei due tempi (t0 e t1) continua ad essere minore del tMin (ovvero la granularità fratto la percentuale di errore dei dati, nel nostro caso 1%). La crescita di *"rip"* è esponenziale. Nel secondo while invece, considerando le ripetizioni, troviamo il tempo minore senza cicli falliti tramite il metodo numerico di bisezione.

"ripCalculateGross" calcola il tempo di esecuzione del programma in totale, compreso del tempo di esecuzione dell'algoritmo, il tempo per il caricamento dei dati e come per *"ripCalculateTare"*, vengono tenuti in considerazione anche i cicli errati che possono essere presenti. In questo caso abbiamo inserito un numero di al più 5 cicli errati.

```
private static long ripCalculateGross(int d, long tMin){
    long t0=0;
    long t1=0;
    long rip=1;
    while(t1-t0 <= tMin) {
        rip = rip*2;
        t0 = System.currentTimeMillis();
        for(int i = 0; i <= rip; i++) {
            execute(prepare(d), d);
        }
        t1= System.currentTimeMillis();
    }
    long max = rip;
    long min = rip / 2 ;
    int cicliErrati = 5 ;

    while (max-min >= cicliErrati) {
        rip = (max+min) / 2 ;
        t0 = System.currentTimeMillis();
        for(int i = 1; i<=rip; i++) {
            execute (prepare(d), d);
        }
        t1= System.currentTimeMillis();
        if(t1-t0 <= tMin) {
            min = rip;
        } else {
            max = rip;
        }
    }
    return max;
}
```

La stessa metodologia viene usata sia in *"ripCalculateTare"* sia in *"ripCalculateGross"*. Nella prima viene considerato il tempo di esecuzione della preparazione dei dati che andranno inseriti nell'algoritmo, mentre in *"ripCalculateGross"* viene considerato tutto il processo, ovvero dalla creazione dei dati allo svolgimento dell'algoritmo.

```
private static ArrayList<Integer> prepare (int d){

    return creatore(d);
}

private static long execute(ArrayList<Integer> array, int d){

    return QuickSelect(array,  inizio: 0,  fine: array.size() - 1,  k: d / 2);
}
```

La funzione *"prepare"* inizializza l'algoritmo che successivamente andrà' ad inserire i dati nell'array per lo svolgimento dell'algoritmo. *"execute*" invece esegue effettivamente l'algoritmo, senza contare come siano stati prodotti i dati in input.

```
private static double mediumNetTime(int d, long tMin) {

    double ripTara = ripCalculateTare(d, tMin);
    double ripLordo = ripCalculateGross(d, tMin);
    long t0 = System.currentTimeMillis();
    for(int i = 1; i<=ripTara; i++){
        prepare(d);
    }
    long t1 = System.currentTimeMillis();
    long tTara = t1-t0;
    t0 = System.currentTimeMillis();
    for(int i = 1; i<=ripLordo; i++){
        execute(prepare(d), d);
    }
    t1 = System.currentTimeMillis();
    double tLordo = t1 - t0;
    double tMedio = (tLordo / ripLordo) - (tTara / ripTara);
    return tMedio;
}
```

In questa parte implementiamo il calcolo del tempo caricando il valore minimo (esecuzione della parte di caricamento dati) e il valore massimo, ovvero sfruttiamo il tempo impiegato dall'algoritmo per calcolare i risultati finali, prendendo in input i dati creati randomicamente.

In “mediumNetTime”, usiamo i vari metodi già realizzati (*“RipCalculateGross”*, *“ripCalculateTare”*, *“prepare”* ed *“execute”).* Con il primo for controlliamo nuovamente il tempo di preparazione dei dati mentre nel secondo for, calcoliamo i tempi dell’esecuzione dell’algoritmo. Tutto ciò serve per calcolare il tempo medio sottraendo i tempi messi a rapporto con i valori precedentemente calcolati.

```
private static double[] misurate(int d, int c, double za,
                                 long tMin, double DELTA) {
    double t=0;
    double sum2 = 0;
    double cn = 0;
    double e;
    double s;
    double delta;
    double m;
    do {
        for(int i=1; i<=c; i++) {
            m = mediumNetTime(d, tMin);
            t+=m;
            sum2 = sum2 +(m*m);
        }
        cn= cn+c;
        e=t/cn;
        s= Math.sqrt((sum2/cn - (e*e)));
        delta= (1/Math.sqrt(cn)) *za *s;
    }
    while(delta>DELTA);
    double[] result = new double[2];
    result[0] = e;
    result[1] = s;
    return result;
}
```

“Misurate” è la parte più importante dell’intero nucleo dell’algoritmo per la misurazione dei tempi, poiché vengono caricate tutte le variabili necessarie per inserire i valori delle ripetizioni, del *“delta”* e dell’errore della ripartizione della gaussiana (za). Tutto ciò serve proprio per collocare in un unico punto tutti gli input necessari alla ricerca dei tempi.
Con il do-while verifichiamo quando l’errore, ovvero la variabile *“DELTA”*, è maggiore del valore inserito in input (variabile *“delta”*). Se i valori superano l’1%, viene caricato il valore successivo. Nel for invece, viene tenuto il conto dei cicli di misurazioni da effettuare, ovvero tutti i risultati del tempo medio, tramite la funzione *“mediumNetTime”*.
In output sono riportate le variabili *“e”* ed *”s”* che sono rispettivamente la media dei tempi e la deviazione standard.

Per calcolare la varianza campionaria abbiamo seguito questa formula. Mettendola sotto radice, troviamo la deviazione standard:

$$s(n)^2 = \frac{1}{c_n} \sum_{i=1}^{c_n} X(n,i)^2 - E'[X(n)]^2$$

Nella formula, la prima parte ($X(n,i)^2$) della sommatoria è rappresentata dalla variabile *"sum2",* la seconda parte (dopo il segno meno), è rappresentata dalla variabile *"e"* al quadrato.

Nel main abbiamo inizializzato le variabili principali, tra cui *"c"* che rappresenta il numero di misurazioni dell'algoritmo. Aumentando il valore, il numero di ripetizioni aumenta e di conseguenza migliora la precisione e aumenta il tempo totale di calcolo dei tempi.
Nel for memorizziamo tutti i tempi nell'array "t". Il "contatore * 2", serve per inserire i dati che verranno trascritti sul foglio Excel. I dati verranno caricati con un ordine tale che il primo dei due risultati sia sempre il tempo e il secondo la deviazione standard. Abbiamo quindi inserito questo contatore solo per ordinare gli elementi dell'array "t".

```
public static void main(String[] args) throws IOException {

    int c = 5;
    double za = 2.32;
    double percent = 0.01;
    long tMin = (long) (Granularita() / percent);
    double DELTA = 0.01;
    int contatore = 0;

    double[] mis;
    double[] t = new double[1000];

    for (int i = 100; i <= 6000000; i = i + ((i * 10) / 100)) {

        mis = misurate(i, c, za, tMin, DELTA);
        if (mis[0] < 1000) {

            t[contatore * 2] = mis[0];
            t[(contatore * 2) + 1] = mis[1];

            contatore++;

        }
    }
```

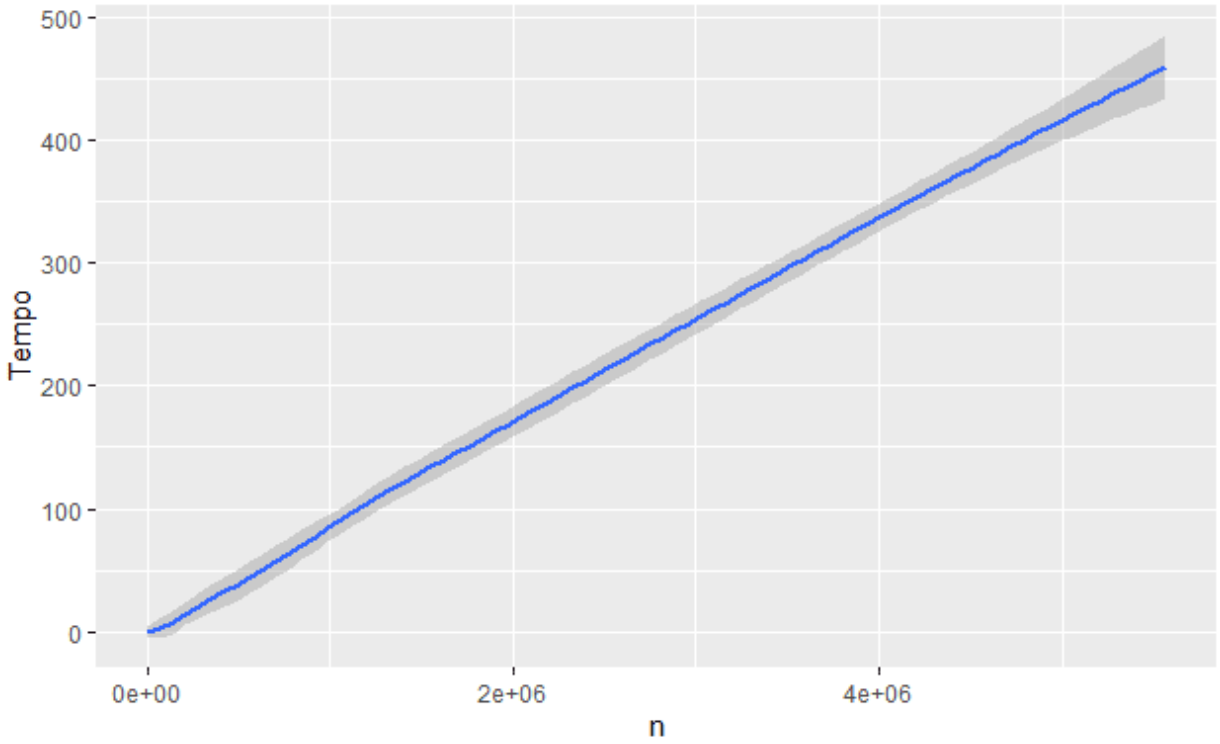
Una volta che l'algoritmo ha finito, importiamo i dati in output direttamente su Excel. La libreria che abbiamo usato per farlo è "hu.blackbelt.bundles.poi:org.apache.poi:4.0.1_1".

```
    XSSFWorkbook workbook = new XSSFWorkbook();
    OutputStream os = new FileOutputStream( name: "TempiHS6M.xlsx");
    Sheet sheet = workbook.createSheet();
    Row row = sheet.createRow( rownum: 1);
    Cell cell = row.createCell( column: 1);
    cell.setCellValue("n");
    Cell cell1 = row.createCell( column: 2);
    cell1.setCellValue("Tempo");
    Cell cell0 = row.createCell( column: 3);
    cell0.setCellValue("delta");
    Cell cell01 = row.createCell( column: 4);
    cell01.setCellValue("sm");
    int cont = 0;
    for (int nn = 100; nn <= 6000000; nn = nn + ((nn * 10) / 100)) {
        Row row1 = sheet.createRow( rownum: cont + 3);
        Cell cell2 = row1.createCell( column: 1);
        cell2.setCellValue(nn);
        Cell cell3 = row1.createCell( column: 2);
        cell3.setCellValue(t[(cont * 3)]);
        Cell cell4 = row1.createCell( column: 3);
        cell4.setCellValue(t[(cont * 3) + 2]);
        Cell cell5 = row1.createCell( column: 4);
        cell5.setCellValue(t[(cont * 3) + 1]);
        cont++;
    }
    workbook.write(os);
}
```

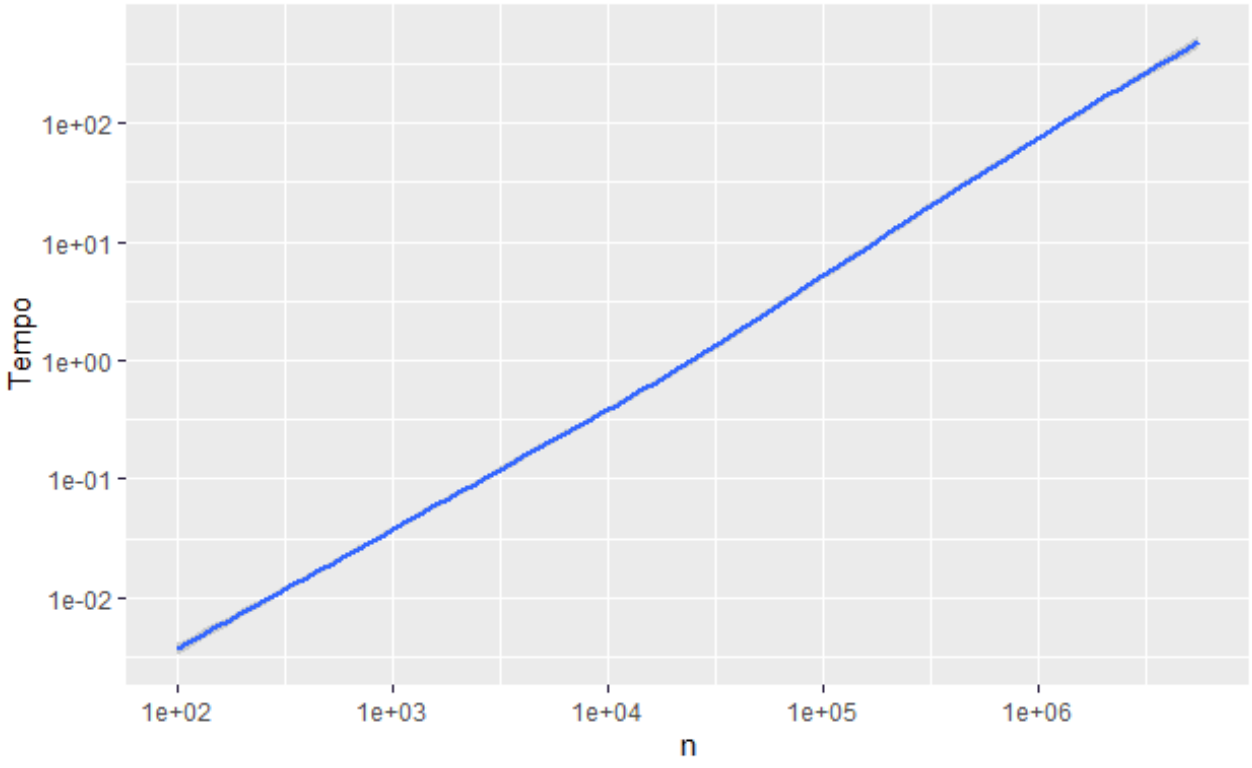
5. Presentazione dei risultati e considerazioni

Per fare i grafici abbiamo usato R. Nei grafici seguenti il Tempo è sempre rappresentato in millisecondi e collocato sull'asse y mentre *"n"* rappresenta la grandezza del vettore ed è collocato sull'asse x. I tempi sono stati presi con un aumento del 10% della lunghezza dell'array (il valore successivo è maggiore del 10% rispetto al precedente). Il limite superiore della grandezza del vettore è stato fissato a 6 milioni di elementi.
Nei grafici la linea in azzurro rappresenta regressione lineare dei dati mentre la parte in grigio esprime l'errore standard.

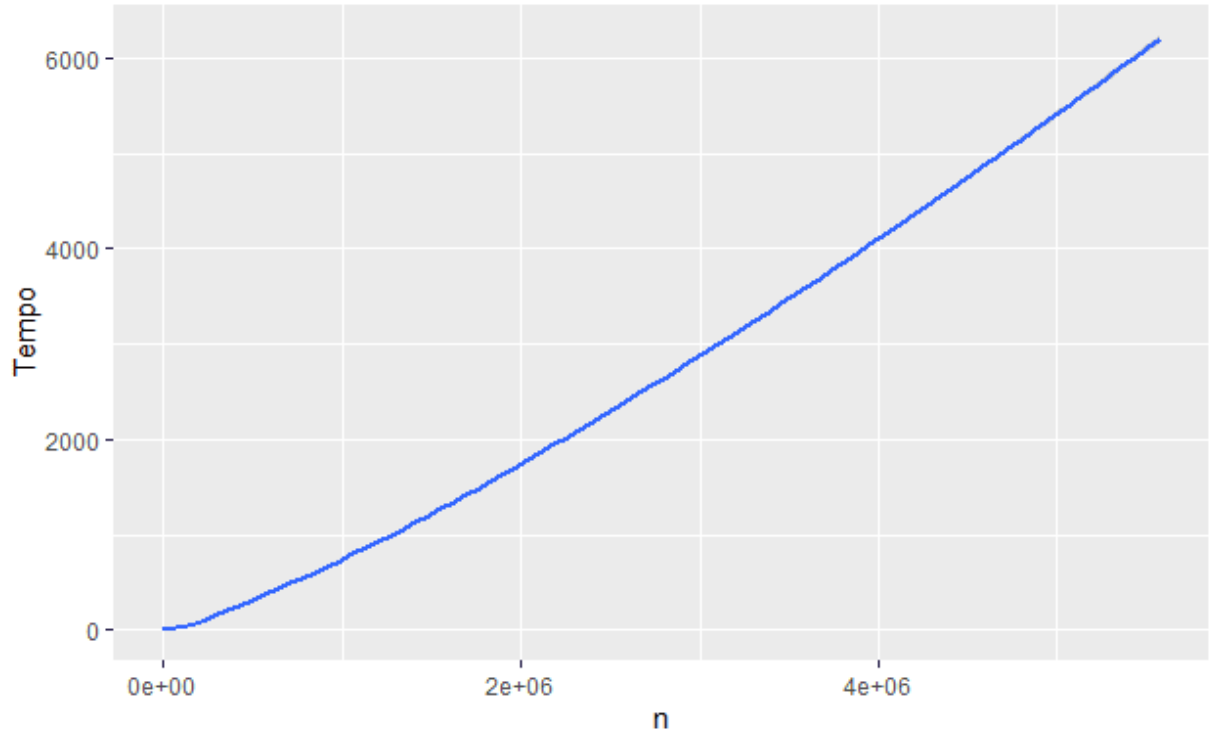
QuickSelect con scala lineare



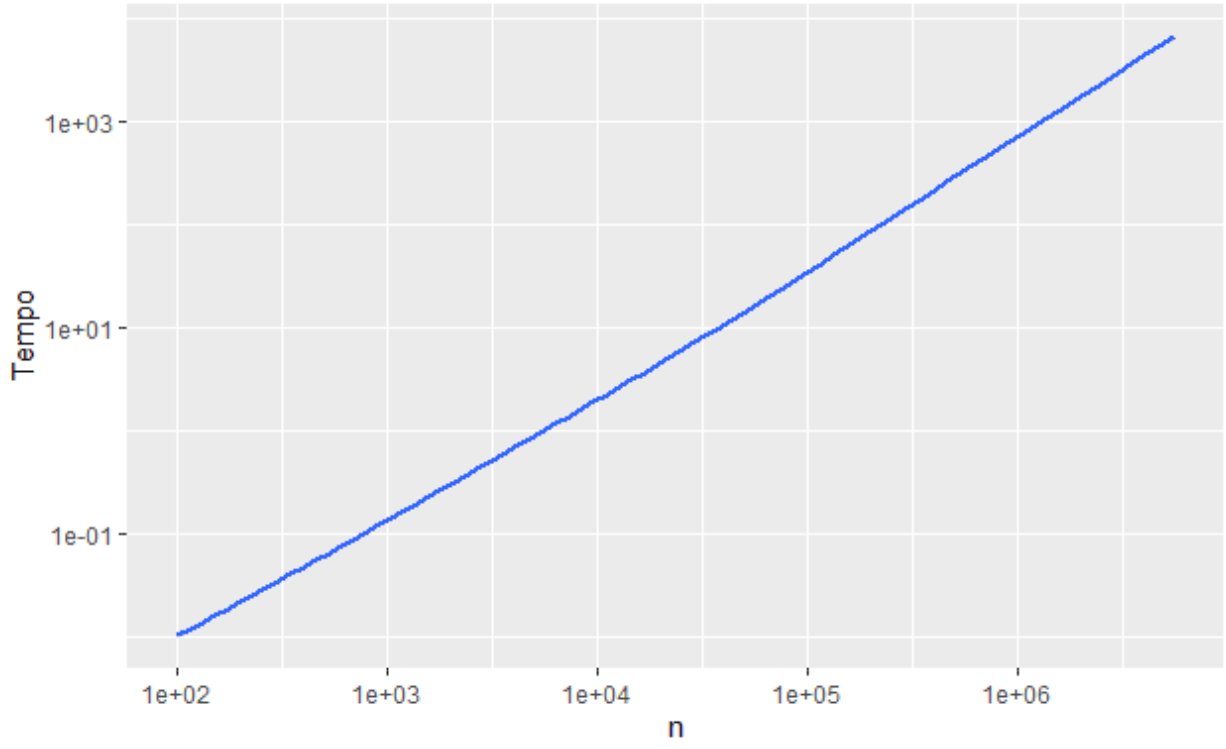
QuickSelect con scala logaritmica



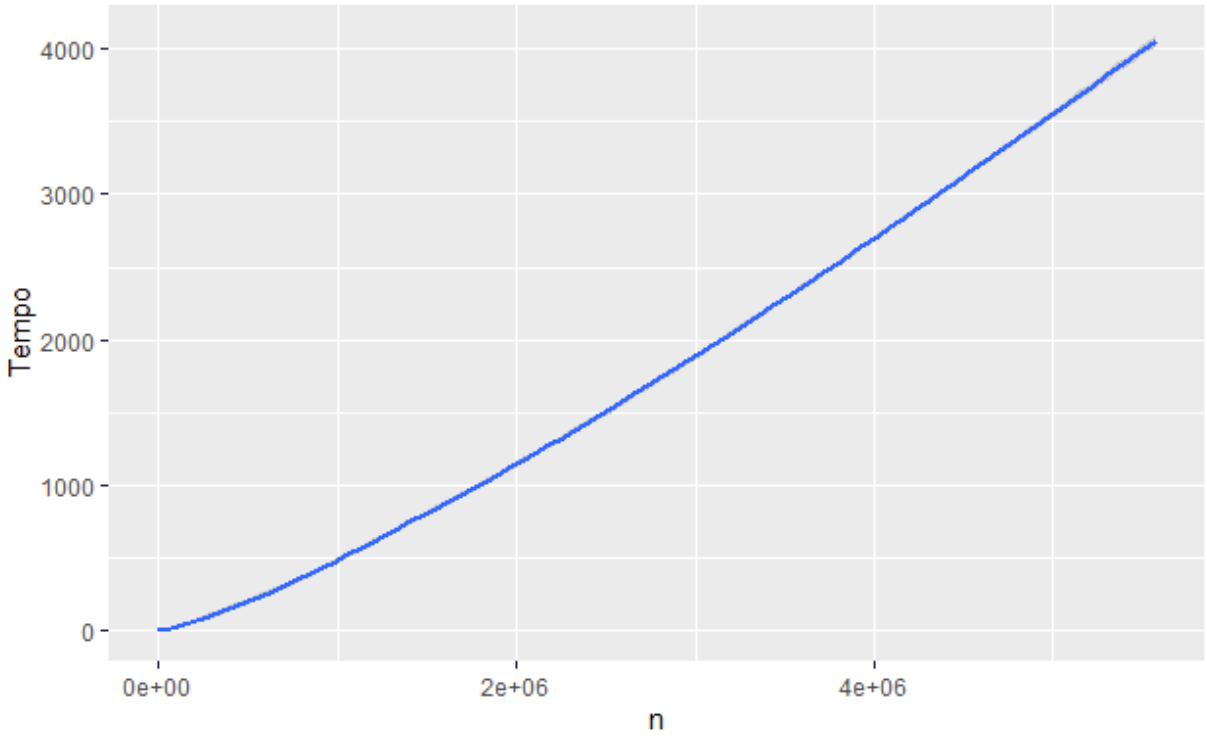
HeapSelect con k = n / 2 e scala lineare



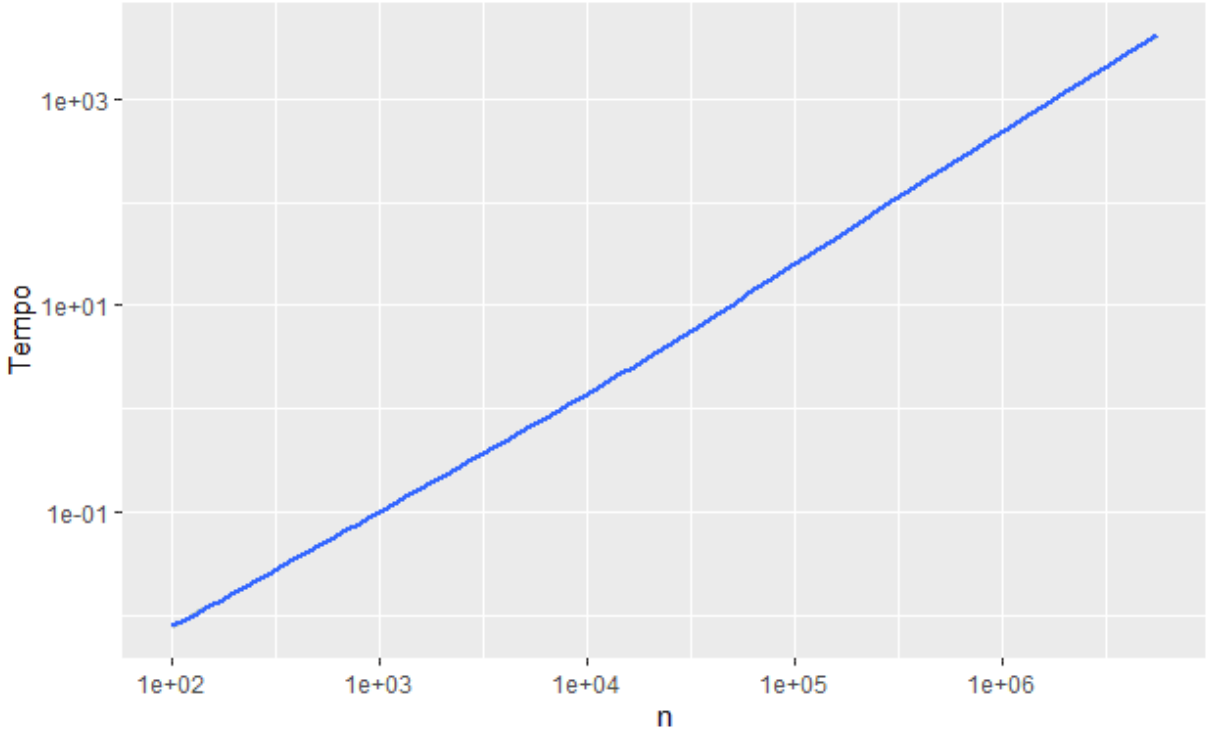
Heap Select con k = n / 2 e scala logaritmica



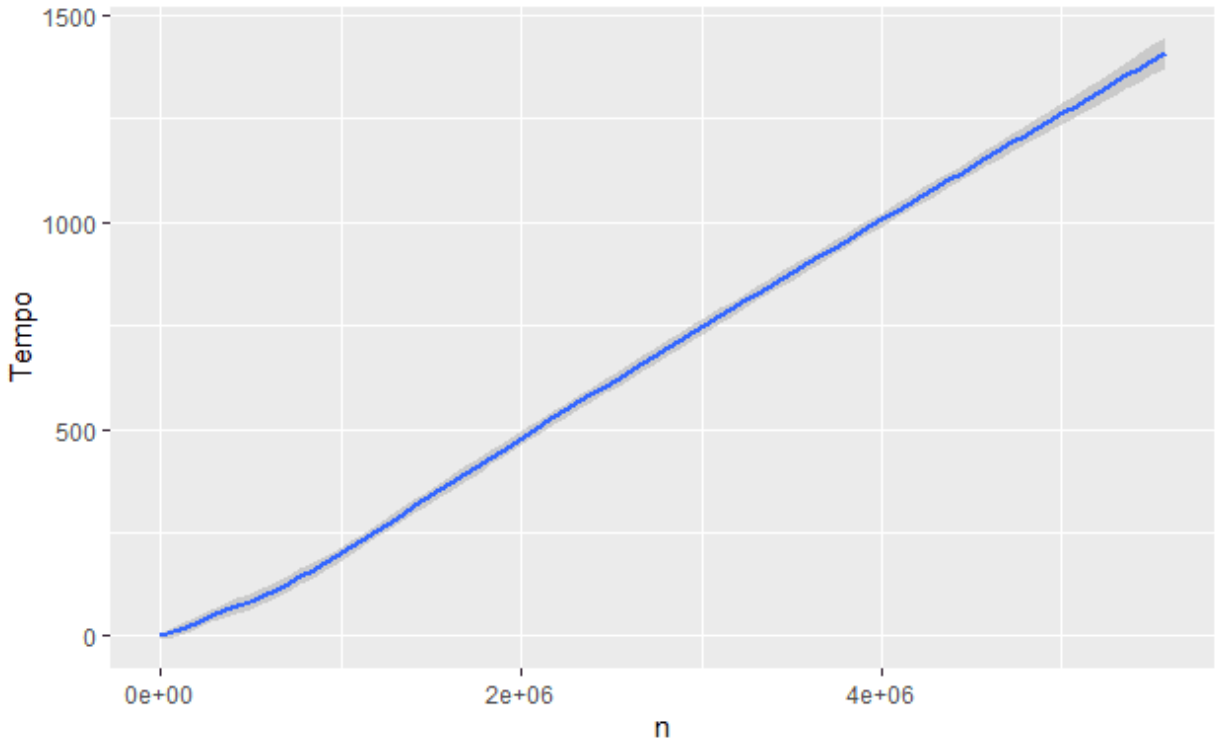
HeapSelect con k = n / 3 e scala lineare



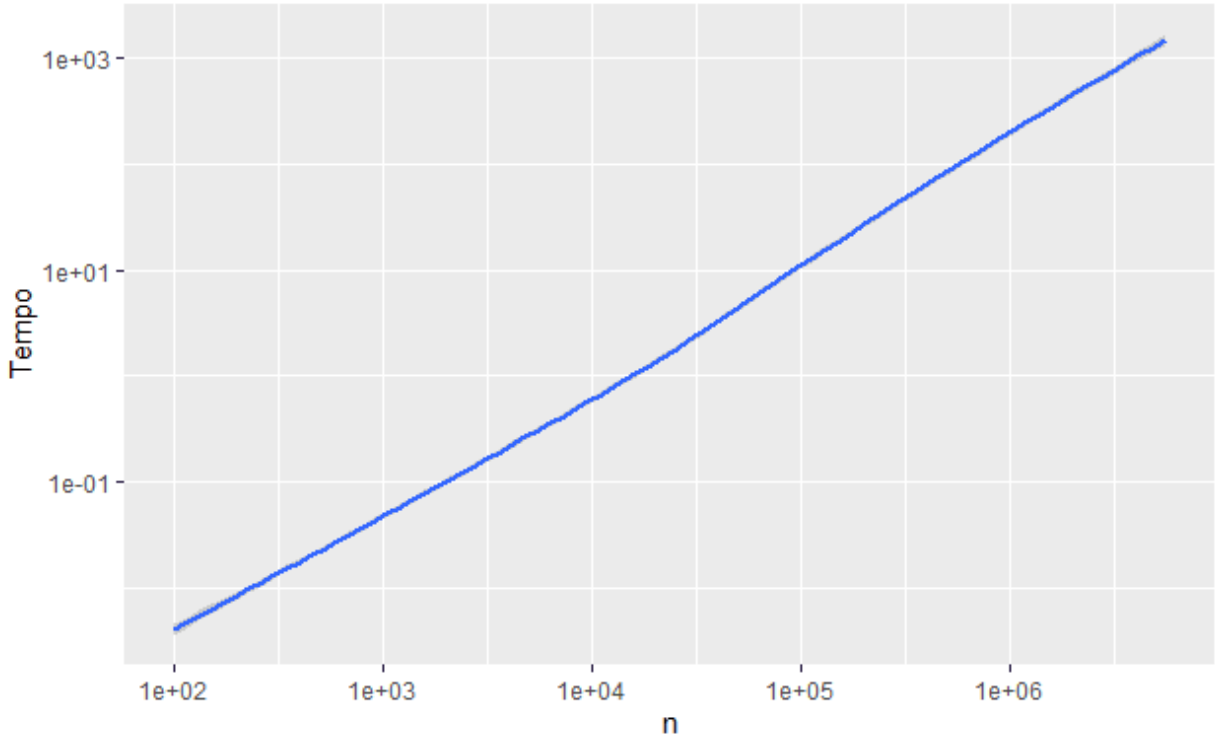
HeapSelect con k = n / 3 con scala logaritmica



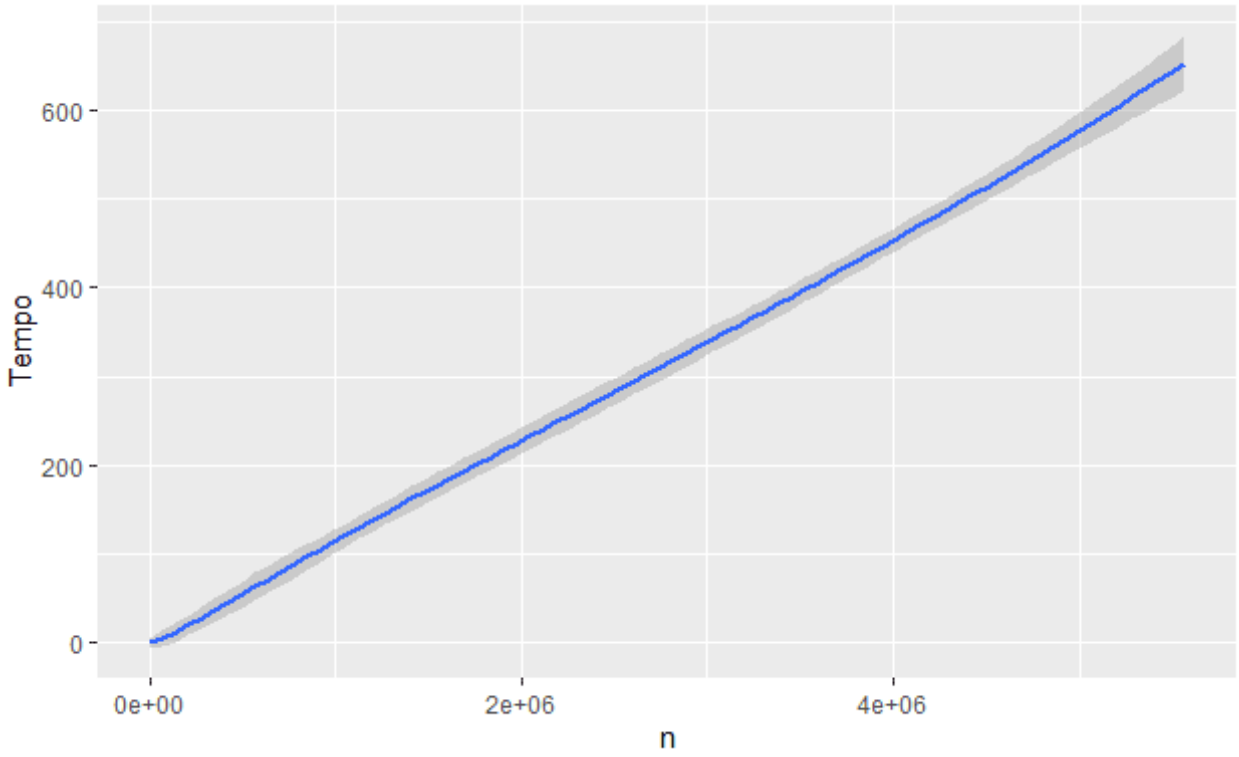
HeapSelect con k = n / 10 e scala lineare



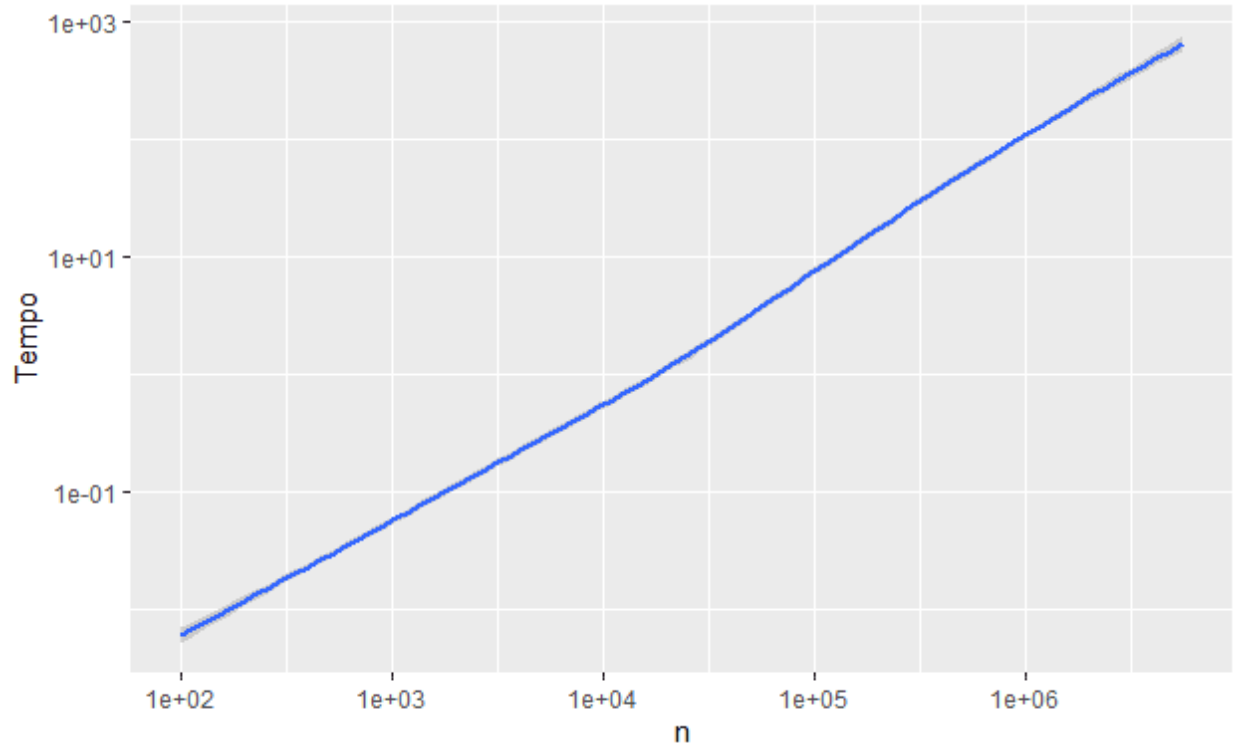
HeapSelect con k = n / 10 con scala logaritmica



Medians Of Medians con scala lineare

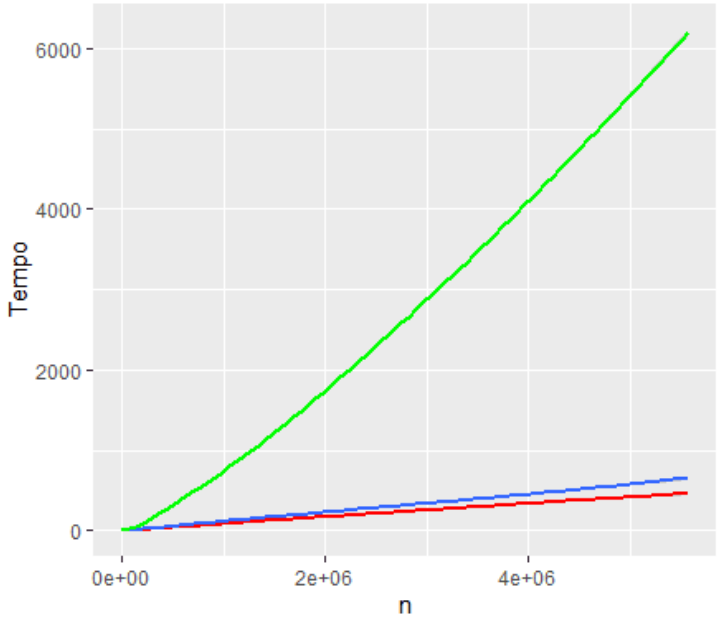


Medians Of Medians con scala logaritmica

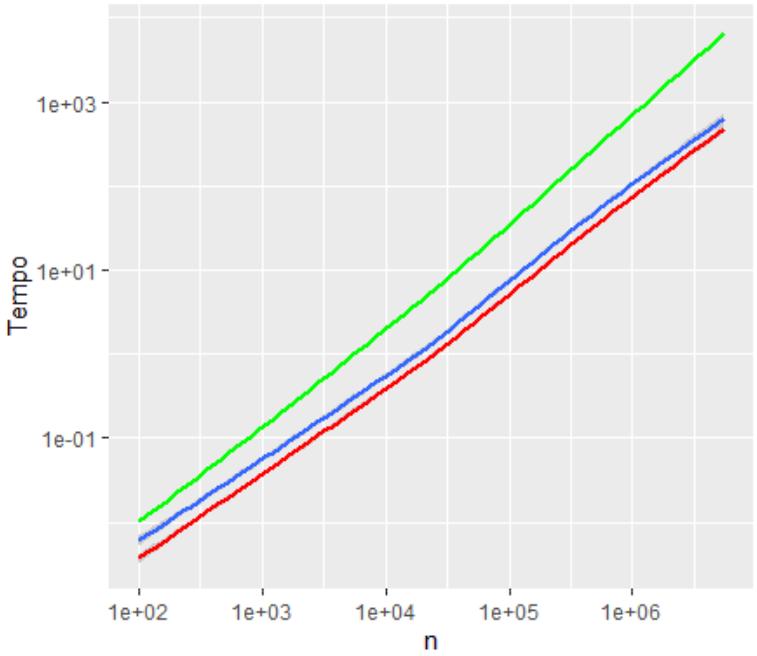


Sostanzialmente i grafici risultano coerenti con le nostre aspettative:
a) QuickSelect ha una complessità temporale di O(n) nel caso medio e di $\Theta(n^2)$ nel caso peggiore. Dal grafico si evince che abbiamo un andamento lineare, il che soddisfa le nostre aspettative sui tempi.
b) HeapSelect ha una complessità temporale di O(n + klogk). I tempi di questo algoritmo variano in base al k. Dai grafici è evidente che all'aumentare di k, aumentano i tempi.
c) MedianSelect ha una complessità temporale di $\Theta(n)$. Dal grafico si può osservare che abbiamo un andamento lineare, il che rispetta le nostre aspettative.

Tre algoritmi in scala lineare



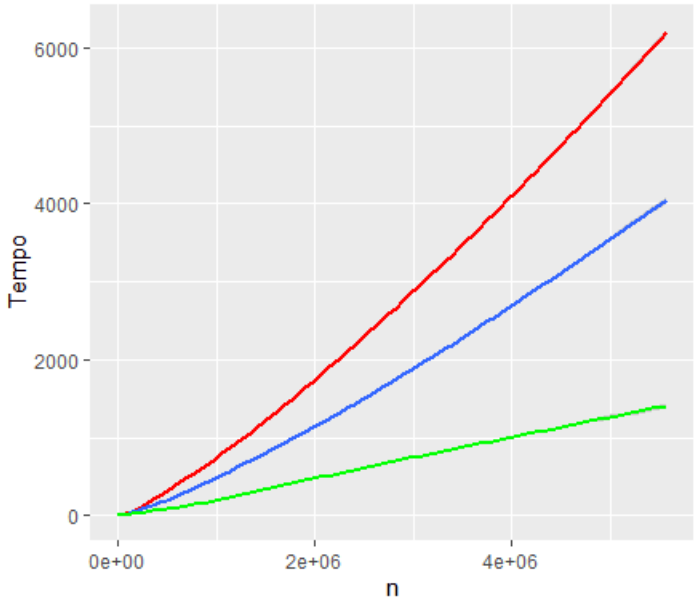
Tre algoritmi su grafico logaritmico



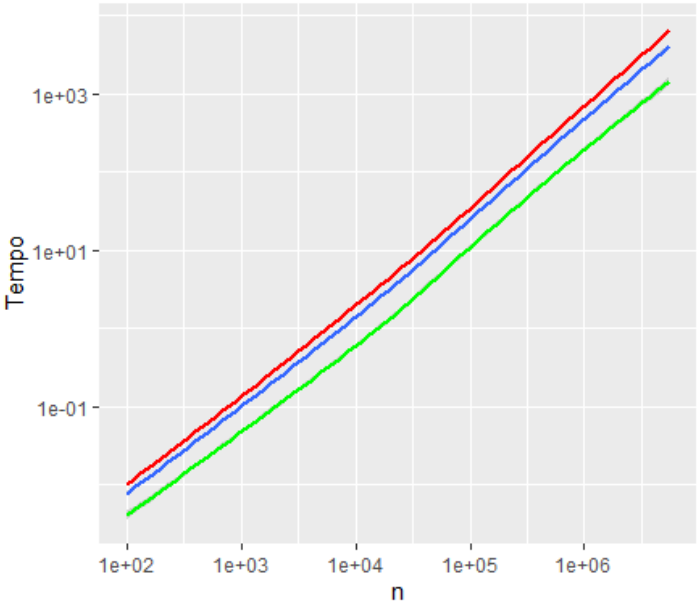
(BLU Medians of Medians, VERDE HeapSelect, ROSSO QuickSelect)

Come si può vedere l'algoritmo più veloce è QuickSelect mentre quello più lento è HeapSelect. QuickSelect e Medians od Medians hanno prestazioni simili fino ad n uguale a un milione circa. HS cresce molto più velocemente degli altri due, perché la creazione della heap impiega un certo tempo per essere costruita. In questo grafico, tutti e tre gli algoritmi sono stati eseguiti con lo stesso k che equivale a n/2. Per questo, dato che il k è in posizione centrale nell'array, i tempi di HS crescono molto in fretta. Se paragonato ad altri HS con k più vicini all'inizio dell'array, si nota che questi ultimi sono più veloci.

Tutti HeapSelect in scala lineare



Tutti HeapSelect su grafico logaritmico



(ROSSO HS con k = n/2, i BLU HS con k = n/3, VERDE HS con k = n/10)