Analisi di tre differenti algoritmi per la risoluzione del problema della selezione

In questa relazione confronteremo le prestazioni di tre diversi algoritmi di selezione.

Il problema della selezione è il seguente:
Dati: un array A non ordinato di n elementi: A[1 … n], un intero k con $1 \leq k \leq n$
Trovare: k-esimo elemento più piccolo appartenente ad A

Gli algoritmi che analizzeremo sono i seguenti:

Nome algoritmo	**Complessità (caso pessimo)**	**Complessità (caso medio)**	**Complessità (caso ottimo)**
Quick Select (QS)	$O(n^2)$	$O(n)$	$O(n)$
Heap Select (HP)	$O(n + k \log_2 k)$	$O(n + k \log_2 k)$	$O(n + k \log_2 k)$
Median of Medians Select (MOMS)	$O(n)$	$O(n)$	$O(n)$

Il valore di k scelto nelle misurazioni sperimentali:
- randomico per QS, MOMS
- costante a n/4 per HP in quanto dipende dal valore di k

Il codice è compilato ed eseguito in c++.
Parti di codice sono state importate dal c (si noti l'uso dei puntatori in alcuni metodi).

Notazione:
A[i … j] indica gli elementi array A dalla posizione i alla j comprese
x % y indica la funzione resto tra x e y
TDEM sta per "tale dimostrazione è omessa"

PARTE UNO: GLI ALGORITMI

QUICK SELECT (QS)

Il quick select si **basa sull'algoritmo partition** e **non necessita** di **strutture dati** esterne.
Solo la forma classica, quella ricorsiva, necessita di memoria, lo stack per le chiamate ricorsive, ma quella qui analizzata è implementata in modo iterativo con un semplice while.

Partition

Il partition funziona nel modo seguente:

> **Dati**: un array A non ordinato di n elementi, un elemento x detto "pivot" $\in$ A
> **Funzionamento**: ordina A in tre parti: valori minori o uguali x, x, i valori maggiori x
> **Output**: ritorna indice i pivot, A[1 … i-1] ≤ pivot, A[i] = pivot, A[i+1 … n] > pivot

Funzionamento

Creiamo e inizializziamo due variabili left = 1, right = n.
Eseguiamo il partition da left a right, su un **pivot casuale** dell'array (che può essere il primo, l'ultimo, quello in posizione n/2, uno randomico, nulla cambia in quanto l'array stesso è composto da numeri casuali).
Confrontiamo indice i restituito dal partition con il parametro k. Abbiamo tre casi:

- i = k: ricerca è terminata, **ritorniamo il valore** trovato di A[i]
- i < k: **elemento** cercato **è maggiore** del pivot: rieseguiamo QS da i+1 a right
- k < i: **elemento** cercato **è minore** del pivot: rieseguiamo QS da left a i-1

Si procede fino a che i non assume lo stesso valore di k.

Analisi caso migliore e peggiore

Dopo ogni partition **array è spezzato in due** e la ricerca di k **procede solo in una delle due parti**, le quali **possono variare di dimensione** in base alla posizione del pivot nell'array.
La scelta ottimale sarebbe il k-esimo stesso, ma esso è il nostro problema non la soluzione.
Una scelta interessante potrebbe essere il mediano da left a right: array viene diviso sempre in due metà uguali, che garantirebbe sempre un O(n). Nonostante ciò QS usa pivot casuale.
Si può notare nella definizione dell'algoritmo stesso una possibile varietà nel numero di iterazioni necessarie per terminare. Infatti tutto dipende dal pivot scelto per il partition:

- **caso migliore:** se pivot casuale è proprio il valore stesso cercato (e ciò accade con una probabilità di 1/n, o anche maggiore se è ripetuto in A) allora dopo una sola esecuzione di partition abbiamo terminato.
- **caso peggiore:** cerchiamo il max(A) e il pivot scelto è, ad ogni iterazione, min(A[left … right]): la partizione di A è pessima: tutto array è a destra del pivot, abbiamo "scartato" solo un elemento ad ogni passo. Per trovare il max abbiamo eseguito θ(n) partition, che ha un costo θ(n), in totale: $O(n^2)$ (lo stesso vale invertendo max e min).

Analisi caso medio

Il caso peggiore di QS in realtà però si realizza molto raramente, e questo implica che la complessità nel tempio medio è O(n), ma TDEM.

Codice

```
int quickSelect(vector<int> vec, int left, int right, int k) {
   while (left < right) {
        int index = partition(&vec[0], left, right - 1);
        if (index == k - 1) {
            return vec[index];
        } else if (index > k - 1) {
            right = index;
        } else {
            left = index + 1;
        }
    }
    return -1;
}
```

HEAP SELECT (HS)

L'heap select è basato sulla **struttura dati ad albero Heap**. Solitamente è implementata come una Min-Heap (nel nostro caso) o una Max-Heap.

Definizione Min-Heap

Una **heap** generica è una **struttura dati** molto semplice. Può essere implementata con un **array A sovradimensionato**, un **intero length** che indica grandezza di A, un **intero size** che indica il numero di elementi nella heap.
I dati si memorizzano in una **min-heap** nel seguente modo:

- **A[1]** è il **minimo** dell'array e radice dell'albero
- **A[2*i]** e **A[2*i + i]** sono **maggiori** di **A[i]**, $\forall i \leq size/2$, sono "figli" di A[i] nell'albero

Quindi in posizione:

- 2*i e 2*i + 1 sono presenti i "figli" dell'elemento in posizione i
- ((i - i % 2) / 2) è presente il "padre" dell'elemento in posizione i

Metodi principali Min-Heap

- **heapify:** risistema albero riportando verso foglie un nodo chiave maggiore suoi figli
  - Costo: O(log n). Caso peggiore parto radice arrivo foglia ⇒ log n passi
- **extractMin**: estrae il minimo dalla heap (sostituisce con ultimo elemento e lo ritorna)
  - Costo: O(log n): scambio A[1], A[size]: θ(1). chiamata a heapify: O(log n)
- **buildMinHeap**: crea la heap: esegue n/2 volte heapify per sistemare array
  - Costo: θ(n): n/2 chiamate a heapify: θ(n/2) * O(log n) =^ θ(n)
  - ^La dimostrazione del costo lineare di build min heap è omessa

Funzionamento

Idea base

Trasformare array in una min heap, θ(n), **estrarne il minimo**, O(log n), **k volte** θ(k).
Costo: θ(n) + O(k * log n) = **O(n + k log n)**.

Ottimizzazione

Utilizzare due min heap: **MH1** e **MH2**.
MH2 è formata formata da **coppie <K, I>** <chiave, indice elemento stesso in MH1>.
MH2 è ordinata secondo K.
Inizialmente:
MH1 viene generata **come sopra**, θ(n); **MH2** inseriamo solo il minimo m di MH1: **<m, 1>**.
Estraiamo il minimo <m, index> da MH2, O(log k).
Inseriamo in MH2 i **figli di m** in MH1:
left: MH1[2 * index], right: MH1[2 * index + 1] (se esistono, ovvero non sforano MH1.size).
Dopo ogni inserimento **left** e **right risalgono MH2** in base alla loro chiave, O(2 * log k).
Il tutto viene **eseguito** sempre **k volte**, θ(k).
Invariante: **MH2.size ≤ k**.
Dimostrazione:
Ad ogni iterazione eseguiamo una rimozione -1 e due inserimenti +2 ⇒ +1 elemento k volte.
Perciò **MH2.size = i+1** dove i è i-esima iterazione $0 \leq i < k$.
Questo implica che risistemazione di MH2 costa O(log k).
Costo: θ(n) + O(k * 3 log k) = **O(n + k log k)**

Analisi

A differenza di QS non esiste un caso migliore, peggiore o medio.
HS è un **algoritmo molto "stabile"** nei tempi di esecuzione: per cercare il k-esimo esegue **k volte** la **l'estrazione del minimo da MH2**, e ognuna di queste costa O(log k).
Il punto fondamentale di HS è che dipende totalmente dal parametro k.
Per ottenere prestazioni accettabili in confronto agli altri algoritmi bisogna:

- usare min-heap per cercare valori "vicino" minimo
- usare max-heap per cercare valori "vicino" massimo

Per cercare valori "vicino" alla mediana non è consigliato:
Se k = n/2 ⇒ Costo = O(n + k log k) = O(n + n/2 log n/2) = O(n + n log n) = O(n log n)
Anche escludendo il θ(n) iniziale per la costruzione dell'heap, comunque da non sottovalutare per array dell'ordine di milioni/miliardi, resta comunque un O(n log n) alquanto lontano dal O(n) del caso medio degli altri due algoritmi.

Codice

```
int callHeapSelect(vector<int> vec, int left, int right, int k) {
    MinHeap h1 = MinHeap(vec, right - 1);
    supportMeanHeap h2;
    h1.buildMinHeap();
    return heapSelect(&h1, &h2, k);
}

int heapSelect(MinHeap* h1, supportMeanHeap* h2, int k) {
    int root = h1->getRoot();
    h2->insert(root, 0);
    for (int i = 0; i < k-1; ++i) {
        int index = h2->extractPos();
        if(h1->leftSon(index) <= h1->heapsize) {
            h2->insert(h1->vec[h1->leftSon(index)], h1->leftSon(index));
            if(h1->rightSon(index) <= h1->heapsize) {
                h2->insert(h1->vec[h1->rightSon(index)], h1->rightSon(index));
    }}}
    int last = h2->nodePos[0].first;
    return last;
}
```

MEDIAN OF MEDIANS SELECT (MOMS)

MOMS è anch'esso **basato sul partition** come QS. La differenza sta nel pivot scelto.
MOMS si impone di garantire anche nel **caso pessimo** un **tempo lineare**.
Per fare ciò ad ogni iterazione prima si **cerca uno "pseudo-mediano" nell'array** e poi lo si usa come **pivot nel partition**, evitando così il caso pessimo del QS.
Il MOMS qui implementato utilizza un array di supporto B, per la ricerca del mediano.
Esistono anche implementazioni in-place.

Ricerca del Mediano dei Mediani

Il mediano è così calcolato:
Si crea intero i = 0, array B di grandezza m = (right - left) / 5 + min((right - left) % 5, 1) ≈ n/5.
Si ordina A[i … i+5] e si copia A[(2i+5) / 2] in B[i / 5]. Si aumenta i di 5 fino a che i > m.
Si noti che il costo dell'ordinamento è costante dato che ogni blocco ha lunghezza costante.
Denominando A[i … i+5] l'i-esimo blocco avremo in B[i] il mediano dell'i-esimo blocco.
Si procede ricorsivamente: B è il "nuovo" A e sarà generato B' di grandezza ≈ m/5 ≈ $n/5^2$
Si termina quando abbiamo trovato il **mediano dei mediani**, ovvero in $n/5^x$ = **$\log_5$ n passi**.

Funzionamento

Si inizializza left = 1, right = n, target = k
Si calcola il **mediano dei mediani x**, come visto sopra, di A[left ... right].
Si esegue **partition** usando come **pivot x**.
Indice **i = target**?:

- Si ⇒ abbiamo trovato valore cercato
- **No** ⇒ si **riesegue tutto** (ricerca mediano e partition) nella **metà interesse**, come QS:
    - si aggiornano left, right ed eventualmente target

Analisi

Ognuna delle operazioni implementata è O(n):

- spezzare array in blocchi da 5 elementi θ(n)
- trovare il mediano di ogni blocco θ(n)
- salvare elementi in B θ(n)
- cercare ricorsivamente il mediano x in B: T(⅕ n)
- eseguire partition usando x come pivot: T(¾ n)

Il "mediano" x trovato è un buon pivot: da ¼ a ¾ elementi sono sia minori sia maggiori di lui.
Quindi l'equazione ricorsiva di complessità è la seguente:
T(n):

- θ(1) se n ≤ 5
- T(⅕ n) + T(¾ n) + θ(n) se n > 5

Dal Lemma:
essendo **⅕ + ¾ < 1** ⇒ le relative equazioni complessità possono essere "ignorate" e si può tenere conto solo del θ(n) ⇒ **è lineare**

Codice

```
int MOMSelect(vector<int> vec, int left, int right, int k) {
    int medianPos;
    bool search = true;
    while(search) {
        int median = getMedianOfMedians(&vec[0], left, right);
        medianPos = partitionPivot(&vec[0], left, right - 1, median);
        int target = medianPos - left + 1;
        if (k < target) {
            right = medianPos;
        } else if (k > target) {
            left = medianPos + 1; k -= target;
        } else {
            search = false;
        }
    }
    return vec[medianPos];
}

#define BLOCK_SIZE 5
int getMedianOfMedians(int *array, int left, int right) {
    if(left >= right - 1) {
        return array[left];
    } else {
        int interval = right - left;
        int bLen = (interval / BLOCK_SIZE) + min(1, interval % BLOCK_SIZE);
        int blocks = 0;
        int B[bLen];
        for(int i = left; i < right; i += BLOCK_SIZE) {
            int limit = min(i + BLOCK_SIZE - 1, right - 1);
            insertionSort(array, i, limit);
            B[blocks++] = array[(i + limit) / 2];
        }
        return getMedianOfMedians(B, 0, bLen);
    }
}
```

PARTE DUE: ESECUZIONE

Come prima cosa prima di poter rilevare il tempo di esecuzione di un dato programma bisogna accertarsi di **garantire** un **errore relativo massimo**. Ovvero il **tempo di esecuzione** misurato **non deve essere minore** di una certa soglia.

STIMA DELL'ERRORE RELATIVO

Vogliamo capire qual è il minor tempo possibile affetto da un errore di una certa soglia ϵ.
Definiamo:
y = tempo esecuzione effettivo
ỹ = tempo esecuzione misurato
R = risoluzione del clock del calcolatore
ε_y = errore relativo di y
ϵ = 0,01 il valore massimo fissato dell'errore relativo
ỹ per definizione è affetto dall'errore della risoluzione:

$$\widetilde{y} \in [y - R, y + R]$$

ε_y per definizione appartiene al seguente intervallo:

$$\varepsilon_y = \frac{(\widetilde{y} - y)}{y} \in [\frac{(y - R - y)}{y}, \frac{(y + R - y)}{y}] = [-\frac{R}{y}, +\frac{R}{y}]$$

Da cui si ricava il valore minimo di y:

$$|\varepsilon_y| \leq \frac{R}{y} \leq \epsilon \Rightarrow y \geq \frac{R}{\epsilon}$$

Aggiungiamo ỹ: la prima disuguaglianza è diretta dalle ipotesi, la seconda viene imposta:

$$y \geq \widetilde{y} - R \geq \frac{R}{\epsilon} \Rightarrow \widetilde{y} \geq \frac{R}{\epsilon} + R$$

Infine sostituendo ϵ con 0,01 si ha:

$$\widetilde{y} \geq \frac{R}{0,01} + R = 100R + R = 101R \Leftrightarrow \widetilde{y} \geq 101R$$

Ora che conosciamo il **minor tempo possibile** affetto da un errore accettato passiamo al calcolo della risoluzione.

CALCOLO DELLA RISOLUZIONE

Per calcolare la risoluzione del calcolatore utilizziamo un **clock costante** (noto anche come anche punto temporale) definito ***steady clock*** all'interno della libreria ***chrono*** di c++.
Ogni clock costante ha due caratteristiche fondamentali:

1. è *monotonico*: i **punti temporali** di questo orologio **non possono diminuire** con l'avanzare del tempo fisico
2. il **tempo** tra un **tick** e l'altro della macchina è **costante**

Questo orologio non è correlato al tempo del classico orologio a muro (per esempio, può misurare il tempo dall'ultimo riavvio), ed è **adatto** per **misurare gli intervalli di tempo** come nel nostro caso.
L'implementazione è la seguente:
Si misura l'ora di inizio ***start* = *now*()**. Si esegue un ciclo ***while* fino** a che l'ora di fine ***end***, continuamente aggiornata a *now*(), rimane **uguale** a ***start***(). Al termine del *while end* sarà

maggiore di *start.* Si calcola la **differenza** tra i due valori da cui si **ottiene** la **risoluzione**, che viene salvata nel vettore di *duration<double>*. Il **calcolo è ripetuto** un centinaio di volte. Alla **fine** si cerca la **mediana dal vettore** delle risoluzioni e la si restituisce.

```
duration<double> resolution() {
    int n = 101;
    vector<duration<double>> res = vector<duration<double>>(n);
    res = resolutionVec(n);
    return quickSelect(res, 0, res.size(), n/2 + 1);
}

vector<duration<double>> resolutionVec(int n) {
    vector<duration<double>> res = vector<duration<double>>(n);
    steady_clock::time_point start, end;
    for (int i = 0; i < n; i++ ) {
        start = steady_clock::now();
        do {
            end = steady_clock::now();
        } while (end == start);
        res[i] = duration_cast<duration<double>>(end - start);
    }
    return res;
}
```

CALCOLO DEI TEMPI DI ESECUZIONE DEI TRE ALGORITMI

Dominio delle possibili lunghezze degli array

Calcolato il tempo di risoluzione, nel nostro caso pari a 1,4 µs, non resta che calcolare i tempi di esecuzione dei tre algoritmi descritti precedentemente. Per cercare di coprire una vasta gamma di possibili dimensioni, abbiamo generato **m = 70 array** di lunghezza n. Partiamo da n[0] = **100 elementi**, e via via li incrementiamo, in modo sparso, **fino** a n[69] = **5 mln** di elementi.

Schema per la misurazione dei tempi di esecuzione

Lo schema di misurazione più intuitivo è il seguente:
1) inizializzare vettore, 2) far partire clock, 3) eseguire algoritmo, 4) fermare clock
ma è assolutamente **sbagliato** in quanto se algoritmo nell'esecuzione impiegasse un tempo paragonabile a quello della risoluzione R ne deriverebbero gravi inconsistenze nelle misure. Abbiamo prima dimostrato che dobbiamo garantire un tempo minimo di 101*R, perciò **l'esecuzione va ripetuta** un numero x di volte per averne la certezza. Lo schema diventa:
1) far partire clock, 2) inizializzare vettore, 3) eseguire algoritmo, 4) tornare punto 2 (x volte), 5) fermare clock

Così si ottiene un tempo "totale" $\mathbf{t_{tot}}$ che è **maggiorato** dei **tempi di inizializzazione** per **x volte**. Per ottenere una misurazione corretta è necessario quindi calcolarsi e salvarsi a priori i tempi di inizializzazione t_{init} dei vari vettori, per scomputarli poi in seguito.
Il tempo esecuzione misurato corrisponde quindi $\mathbf{t_{exec} = (t_{tot} - t_{init}) / x}$.

```
//measure time needed allocate vector[nElements] randomly, repetitions times
duration<double> initializeTime(int nElements, int repetitions) {
    steady_clock::time_point start, end;
    start = steady_clock::now();
    vector<int> vec;
    for (int i = 0; i < repetitions; i++) {
        vec = randomize(nElements); //vec[j] = rand() for j = 0 to nElem
    }
    end = steady_clock::now();
    return (duration<double>)((end - start) / repetitions);
}
```

Schema per la misurazione della varianza (std) dei tempi di esecuzione

Ora che sappiamo **misurare il tempo medio** di un'esecuzione, anche per array molto piccoli, possiamo semplicemente **iterare varie volte** la procedura per ottenere i diversi risultati da cui **calcolare la std dell'algoritmo**. Abbiamo convenuto sufficiente ottenere **20 tempi diversi**. La std la si ottiene direttamente dalla formula:

$$\sum_{i=1}^{n} \sqrt{\frac{|t_i - t_{medio}|^2}{n}}$$

Ora che però non abbiamo più solo un tempo medio ma ne abbiamo 20, dobbiamo scegliere un valore da assegnare al tempo medio.
Prendendo: il minimo → caso ottimo dell'algoritmo.
Prendendo: la media → caso medio dell'algoritmo (che è quello che prendiamo noi).
Prendendo: il massimo → caso peggiore dell'algoritmo.
Oppure si potrebbe prendere la mediana, o solo un valore a caso, o il primo ecc.
Lo schema di esecuzione resta uguale a prima aggiungendo due punti alla fine:
6) tornare punto 1 (20 volte) 7) calcola e salva media e varianza

Codice

```
#define HEAP 0          #define QUICK 1          #define MOM 2
duration<double> res;    #define vdd vector<duration<double>>
int main() {
    nOfArrays = calcNumOfArrays(startingLength);
    res = resolution(); //export data to resolution file
    vdd tinit = initialization(); //export data to init file
    execution(tinit, QUICK, &quickSelect); //export data to quick file
    execution(tinit, MOM, &MOMSelect); //export data to mom file
    execution(tinit, HEAP, &callHeapSelect); //export data to heap file
    return 0;
}
```

```
#define startingLength 100      #define startingNumTimes 150
#define nExecSTD 20             #define RELATIVE_ERROR 0.01
void execution(vdd tinit, int type, int (*function)(vector<int>,int,int,int)){
    int nElements = startingLength; //start with 100 elements
    int nTimes = startingNumTimes; //start repeating execution 150 times
    vdd texec(nOfArrays); //declare vector[70]: save exec time
    vector<double> std(nOfArrays); //declare vector[70]: save std time
    steady_clock::time_point start, end;
    srand(time(NULL));
    for(int i = 0; i < nOfArrays; i++) {
        vdd ttoti(nExecSTD); //declare vector[20]: calc medium exec, std time
        for(int h = 0; h < nExecSTD; h++) {
            int k = type == HEAP ? nElements / 4 : rand() % nElements + 1;
            start = steady_clock::now();
            vector<int> vec;
            for(int j = 0; j < nTimes; j++) {
                vec.clear();
                vec = randomize(nElements); //vec[j] = rand() for j=0 to nElem
               //invoke is functional method to execute a parameterized method
                std::__invoke(function, vec, 0, nElements, k);
            }
            end = steady_clock::now();
            ttoti[h] = (duration<double>)((end - start) / nTimes);
        }
        for(int j = 0; j < nExecSTD; j++) {
             ttoti[j] -= tinit[i]; //ttoti becomes texeci subtracted init time
        }
        texec[i] = (duration<double>) mean(ttoti); //get medium time of ttoti
        std[i] = meanSquaredError(ttoti); //get std time of ttoti
        double measuredError = texec[i].count() * nTimes;
        double relativeError = res.count() / RELATIVE_ERROR + res.count();
        if(measuredError < relativeError) {
            i--; //ỹ < 101R: have to recalculate everything of i-iteration
        } else { //everything is ok, can update nTimes, nElements
            nTimes = updateNumOfTimes(nTimes); //decrease it
            nElements = updateNumOfElem(nElements); //increase it
        }
    }
    printToFile(texec, std, type); //export to file to save data collected
}
```

PARTE TRE: ANALISI DATI

Questo è un riassunto dei dati raccolti: per il tempo di esecuzione, il valore della varianza e la percentuale tra essi di ogni algoritmo ne sono elencati il valore massimo, mediano, minimo, medio e la varianza.

DATI OTTENUTI, IN BREVE

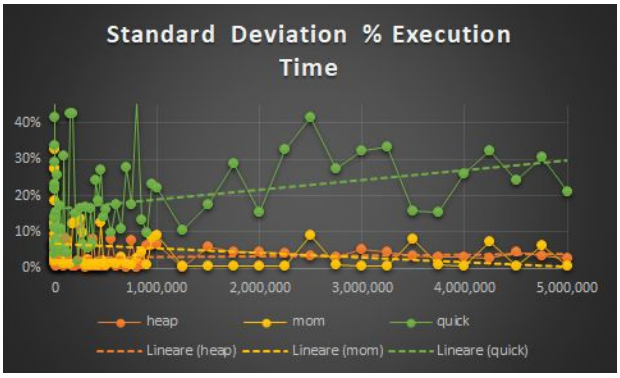
	n° elem	heap select			mom select			quick select		
		time exec	std	std % t exec	time exec	std	std % t exec	time exec	std	std % t exec
max	5.000.000	1,57E+00	6,18E-02	11,42%	2,42E-01	1,51E-02	32,83%	1,12E-01	3,21E-02	54,19%
median	287.500	6,81E-02	5,34E-04	2,56%	1,40E-02	2,32E-04	2,67%	6,20E-03	6,37E-04	16,45%
min	100	1,16E-05	8,35E-07	0,28%	2,80E-06	7,69E-07	0,46%	1,28E-06	5,16E-07	1,88%
mean	904.993	2,64E-01	9,33E-03	3,06%	4,41E-02	1,33E-03	5,51%	2,02E-02	4,81E-03	18,88%
std	1.363.050	4,22E-01	1,56E-02	2,65%	6,67E-02	3,27E-03	6,68%	3,04E-02	8,30E-03	11,93%

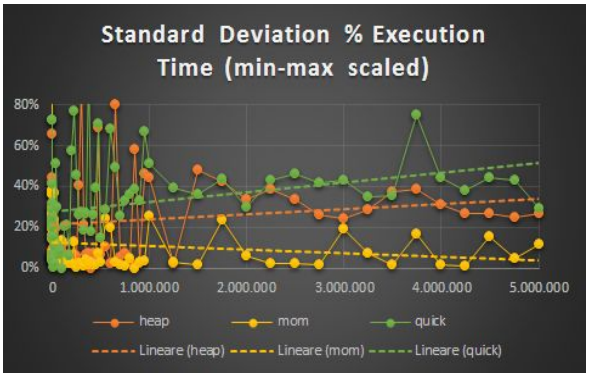
IL TEMPO DI ESECUZIONE

Il primo dato che si guarda quando si comparano gli algoritmi è il tempo di esecuzione.
In ogni grafico ogni algoritmo ha: i suoi valori “reali”, le relative linee di tendenza.

I due grafici **confermano le complessità studiate**: dal primo si nota come **HS**, non lineare, impieghi **più tempo** di **QS e MOMS**, mentre il secondo, con gli assi logaritmici, ne conferma la complessità asintotica: **all’aumentare di n** le **curve rallentano**, ovvero tendono ad assumere la forma di rette parallele all’asse delle x, sia per i due algoritmi lineari sia per HS, “quasi lineare” per k sufficientemente piccolo. Questo è dovuto dal fatto che i dati **non sono quadratici**, altrimenti avremmo ancora delle rette. Non sorprende invece che **QS** sia **più veloce** di **MOMS**: nonostante un $O(n^2)$ nel caso pessimo QS è il più veloce. **A fare la differenza** nei tempi è proprio la **ricerca del pivot per MOMS**: garantisce sempre la linearità, ma ciò incide sui tempi di esecuzione.

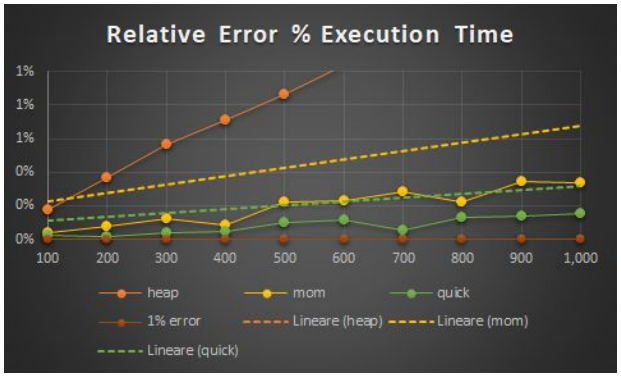
LA VARIANZA (STD)

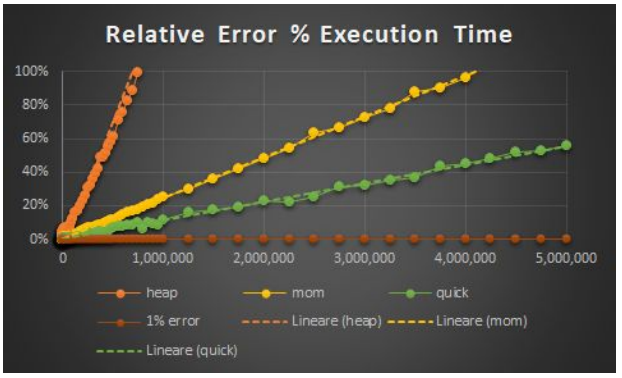




La **varianza**, o standard deviation, è un indice statistico con cui notare come variano i dati. La abbiamo utilizzata per dedurre la **"stabilità" dei tempi di esecuzione** di un dato algoritmo, sullo stesso numero di elementi, ma con valori diversi al suo interno. In pratica si analizza se il tempo di esecuzione di un algoritmo **dipende dall'array di input**. Questo è vero per gli algoritmi che hanno tempi nei casi migliori e peggiori diversi.
La **min-max normalization** invece è utile per analizzare dei **dati** i cui valori tra di essi non sono "vicini", e li rapporta tutti **in scala nell'intervallo [0, 1]**.
QS, che può variare da $O(n^2)$ a O(n), ha **varianza: media alta, del 18,9%**; e con un valore ≥ 20% nel 40% dei casi. Una cosa interessante è che spesso negli algoritmi la std è alta con array piccoli, e cala all'aumentare di n: per la legge dei grandi numeri è più difficile realizzare degli improbabili casi ottimi/pessimi. Ma questo non vale per QS: con **n >= 1,5 mln**, la **varianza** minima è del 15%, la **media del 26%**. Infatti nel grafico si vede la **linea di tendenza crescente** per QS. Questo è dovuto dal fatto che più n è grande più la probabilità di trovare subito il k-esimo al 1° tentativo cala drasticamente (limite per $x \to \inf$ di $1/x = 0$).
MOMS invece funziona esattamente come accennato prima: al crescere di n la std cala. Infatti la std media: per n < 1k è del 17%, gli unici tre casi che sono il sopra il 25% sono con n ≤ 400. Tutti gli altri sono sotto il 15% (tranne un caso) per n > 2k è solo del 3,61%. Questo perchè con la ricerca dello pseudo-mediano **si ottiene sempre un pivot abbastanza centrale**, (uniformando i tempi di esecuzione) ma, come visto prima, il caso ottimo si ha sempre più raramente. Questo implica che la **linea di tendenza cali**. Avendo un caso pessimo, medio e ottimo tutti uguali ci si attende una **std media bassa**, e così è: **5,51%**.
HS: sappiamo già che dipende input, non dall'array ma da k. Tenendo un k costante però anche in questo caso ci si aspetta **varianza media bassa**, e infatti è del **3,06%**. HS è ancora più stabile temporalmente di MOMS. Questo perché non c'è nessun sottoalgoritmo con casi più o meno fortunati. HS estrae semplicemente il minimo k volte. Heapify è l'unico metodo in HS che potrebbe variare, ma dato che: i valori erano già in una heap e la heap su cui viene eseguito è quella di supporto, che ha sempre una lunghezza ≤ k, è difficile che ciò accada. Lo dimostrano le **nove sole volte**, il 13% casi, in cui compare una **std media ≥ 7%**.
Dal grafico min-max si nota un fatto quantomeno curioso: la **linea di tendenza** è in **leggero crescendo** ovvero la std media non cala all'aumentare di n. Infatti è alta per valori piccoli: per n ≤ 1k è 4,03%, poi cala per 1k < n ≤ 700 k a 2.29% e infine, **sorprendentemente**, però **cresce** per **n ≥ 750 k** a **3,94%**. Inoltre si può vedere un agglomerato di puntini arancioni in [0, 1 mln] sullo 0% circa, poi invece in tutti i casi per 900 k ≤ n ≤ 5 mln si ha un valore tra il 20% e 60% della std massima (tranne un caso con n = 1,25 mln).

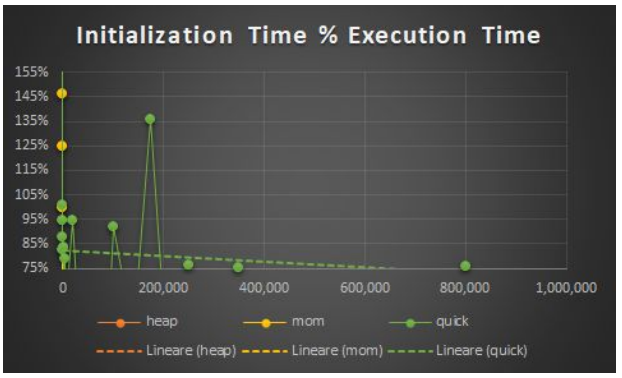
ERRORE RELATIVO





L'errore relativo abbiamo già visto come lo rispettiamo. Nel grafico di sinistra, che è uno zoom per n ≤ 1000, si vede come che l'**errore** è **sempre maggiore** del **minimo necessario**. Il grafico di destra ricalca il grafico sui tempi di esecuzione: ovvio è che maggiore è il tempo esecuzione maggiore è la differenza tra errore minimo da garantire e l'errore effettivo. Ecco perché anche nel grafico degli errori HS cresce più di MOMS che cresce più di QS.

TEMPI INIZIALIZZAZIONE



Come ultima considerazione volevamo solo notare anche la presenza di un **altro errore**. Non è un errore di misurazione bensì di un errore **algoritmico**: quando al tempo di esecuzione totale si **toglie** il tempo di **inizializzazione** per **ottenere** il tempo di **esecuzione** effettivo, si **ottiene un errore**. Diventa consistente se il tempo inizializzazione è dell'ordine di quello di esecuzione. Ma **possiamo considerarlo un errore accettabile** in quanto comunque è garantito l'errore relativo e di conseguenza l'errore nella differenza si può ignorare. Ciò non toglie che il QS è così veloce da produrre ha un rapporto > 80% in 13 casi su 70 (di cui 7 maggiori del 100%), tutti con n < 200 k. Infine anche qua si notano le velocità dei tre algoritmi: più il tempo di esecuzione è minore, più il rapporto aumenta; essendo il tempo inizializzazione costante per i tre algoritmi.

Fonti:
Algoritmi e Strutture Dati (corso presso Università di Udine)
Strutture di Dati e Algoritmi (libro di Crescenzi, Gambosi, Grossi, Rossi)

Repository completa: https://github.com/massimilianobaldo/ProjectASD

143095 Francesco Zuccato
142296 Massimiliano Baldo