Algoritmi randomizzati

> **Note**
>
> Gli algoritmi randomizzati sono algoritmi che effettuano decisioni randomiche durante la loro esecuzione. In pratica un algoritmo randomizzato userà valori generati randomicamente per decidere il cosa fare al prossimo step.

Gli algoritmi randomizzati sono più rapidi dei soliti algoritmi deterministici e anche più facili da implementare. Tutto questo però ha un costo: la risposta può avere la probabilità di essere incorretta.

Applicazione: verificare l'identità di polinomi

Supponiamo di avere due polinomi, $F(x)$ e $G(x)$, dove $F(x)$ è dato come prodotto di fattori e $G(x)$ è dato in forma canonica:

$$[F(x)]: \ (x+1)(x-2)(x+3)(x-4)(x+5)(x-6) \equiv [G(x)]: \ x^6 - 7x^3 + 25$$

Una soluzione banale è quella di portare $F(x)$ in forma canonica e verificare che i coefficienti delle forme canoniche siano uguali. Però questa soluzione ha un problema, ovvero, se notiamo con d il massimo grado del polinomio, allora, per trasformare $F(x)$ nella sua forma canonica impieghiamo $\theta(d^2)$.

Proviamo adesso ad aggiungere un pò di randomness. L'algoritmo ora sceglie un intero r in modo randomico da un intervallo $\{1, \dots, d100\}$, con distribuzione uniforme, ovvero ogni intero ha probabilità equa di essere scelto. Dopodiché l'algortimo computa $F(r)$ e $G(r)$ in tempo $O(d)$, che è decisamente minore rispetto a $\theta(d^2)$. Infine l'algoritmo decide che $F(x) \equiv G(x)$ se $F(r) = G(x)$ oppure $F(x) \not\equiv G(x)$ se $F(r) \neq G(x)$.

Può però succedere che l'algoritmo dia una risposta sbagliata, analizziamo ora i due casi:

- Se $F(x) \equiv G(x)$ allora l'algoritmo ritorna la risposta corretta per ogni r.
- Se $F(x) \equiv G(x)$ e $F(r) \neq G(r)$ allora l'algoritmo ritorna una risposta corretta. Dunque quando l'algoritmo decide che due polinomi sono diversi possiamo essere sicuri che la risposta è sempre corretta.

- Se $F(x) \equiv G(x)$ e $F(r) = G(r)$ allora l' algoritmo ritorna una risposta sbagliata, in altre parole è possibile che l'algortimo decida che due polinomi siano uguali quando in realtà sono divesi. Questo può accadere quando il valore di r corrisponde ad una delle d radici (per il teorema fondamentale dell'algebra, un polinomio di grado d ha al più "d" radici) dell'equazione $F(x) - G(x) = 0$.

Ora però ci resta che analizzare la probabilità di errore dell'algoritmo. Sapendo che $r \in \{1, 2, \ldots, 100d\}$ allora $Pr[err] \leq \frac{d}{100d} \leq \frac{1}{100}$, dunque la probabilità di errore è al piu 1%.

Possiamo infine ridurre la probabilità di erroe al costo di aumentare il numero di esecuzioni dell'algoritmo sulla stessa istanza (lo vedremo meglio dopo).

L'analisi effettuata fino adesso è stata sì un'analisi probabilistica ma non dal punto di vista matematico. Andiamo ora ad analizzare gli algoritmo randomizzati usando gli assiomi, i teoremi e le proprietà del calcolo probabilistico.

Alcune nozioni di probabilità (1)

> **Note**
>
> Uno spazio di probabilità è un concetto usato per modellare situazioni in cui si verificano eventi casuali ed è composto da 3 componenti:
>
> - **Spazio degli eventi** Ω: È l'insieme di tutti i possibili risultati di un esperimento casuale. Ogni singolo risultato è chiamato evento elementare.
> - Una famiglia di insiemi $\mathbb{F}$ di **eventi**, dove ciascun insieme in $\mathbb{F}$ è un sottoinsieme di Ω.
> - Una **funzione di probabilità** $Pr : \mathbb{F} \to \mathbb{R}$ che assegna a ciascun evento un valore tra 0 e 1, rispettando tre assiomi fondamentali:
>   - Per ogni evento E, $0 \leq Pr(E) \leq 1$;
>   - $Pr(\Omega) = 1$;
>   - Per ogni sequenza finita o numerabile infinita di eventi mutualmente disgiunti due a due, $E_1, E_2, E_3, \ldots$,
>
> $$Pr(\bigcup_{i=1} E_i) = \sum_{i \geq 1} Pr(E_i)$$

Analisi dell'algoritmo

Usando queste nozioni, possiamo analizzare il nostro algortimo in maniera più formale e precisa. Abbiamo dimostrato che l'unico caso in cui l'algoritmo da una risposta sbagliata è quando $F(x)$ e $G(x)$ non sono equivalenti e l'algoritmo sceglie randomicamente una radice del polinomio differenza.

- Sia E l'evento che rappresenta il fallimento dell'algoritmo nel dare una risposta corretta.
- Gli elementi dell'insieme E sono le radici del polinomio $F(x) - G(x) \in \{1, \dots 100d\}$.

$$Pr(E) \leq \frac{d}{100d} = \frac{1}{100}$$

  dove d rappresenta il numero di casi favorevoli e $100d$ il numero di casi possibili.

Dunque l'algoritmo ha una probabilità di errore pari a 1%, e dunque sembrerebbe insolito avere un algoritmo che può sbagliare. Negli algoritmi deterministici di solito c'è un tradeoff tra complessità spaziale e complessità temporale. Nell'algoritmo randomizzato che abbiamo visto ci offre un tradeoff tra correttezza e velocità. L'algoritmo ha un precisione dell 99% ma possiamo migliorare questa probabilità?

- Un modo è quello di scegliere r da un insieme di numeri maggiori, per esempio $\{1, \dots, 1000d\}$ così la probabilità di errore è $\leq \frac{1}{1000}$, ma non è sempre possibile fare questa cosa.
- Un altro approccio è di ripetere l' algoritmo molteplici volte sulla stessa istanza sfruttando la proprietà che l'algoritmo ha un **one-sided-error**, ovvero l'algoritmo può sbagliare solo quando decide che i due polinomi sono equivalenti.

Assumiamo di ripetere l'algoritmo k volte.

- Un evento è una sequenza di k scelte $r_1, \dots, r_k$;
- L'insieme dei casi possibili sono tutte le r sequenze nel range $\{1, \dots, 100d\}$;
- La probilità che accada un evento è $(\frac{1}{100d})^k$.
- L'evento **cattivo** corrisponde alla scelta di k radici del polinomio differenza, dunque ci sono al più d^k possibili eventi di questo tipo.
  Dunque, la probabilità che accada un **evento cattivo** è $\leq d^k(\frac{1}{100d})^k$.

Alcune nozioni di probabilità (2)

> **Note**

Due eventi E e F sono **indipendenti** se e soltanto se $Pr(E \cap F) = Pr(E) * Pr(F)$. Volendo generalizzare, gli eventi $E_1, E_2, \ldots, E_k$ sono mutualmente indipendenti tra di loro se soltanto se per ogni sottoinsieme $I \subset [1, k]$,

$$Pr(\bigcap_{i \in I} E_i) = \Pi_{i \in I} Pr(E_i)$$

Dunque, ritornano all'analisi di prima, la probabilità di scegliere una radice singola è $\leq \frac{d}{100d}$, poiché le singole scelte sono **indipendenti** tra loro, otteniamo che, scegliere k radici indipendeti una dopo l'altra ha probabilità $\leq (\frac{d}{100d})^k$.

Note

La probabilità condizionata che si verifichi un evento E sapendo che si è verificato l'evento F è

$$Pr(E|F) = \frac{Pr(E \cap F)}{Pr(F)}$$

Ovviamente, la probabilità condizionata è ben definita se soltanto se $Pr(F) > 0$.

Intuitivamente stiamo cercando la probabilità dell'intersezione dei due eventi ($E \cap F$) all'interno dell'insieme dei eventi definiti da F. Poiché F definisce definisce il nostro spazio di probabilità ridotto, dobbiamo **normalizzare** la probabilità, dividendo per $Pr(F)$ garantendo che la somma delle probabilità condizionate all'interno di F sia 1.

Alcune identità utili:

- $Pr(A|B) = \frac{Pr(A \cap B)}{Pr(B)}$;
- $Pr(A \cap B) = Pr(A|B)Pr(B)$;
- $Pr(A \cap B \cap C) = Pr(A|B \cap C)Pr(B \cap C) = Pr(A|B \cap C)Pr(B|C)Pr(C)$.

Volendo generalizzare, siano $A_1, \ldots, A_n$ una sequenza di eventi. Sia $E_i = \bigcap_{j=1}^{i} A_j$, allora

$$Pr(E_n) = Pr(A_n|E_{n-1})Pr(E_{n-1}) = Pr(A_n|E_{n-1})Pr(A_{n-1}|E_{n-2}) \ldots Pr(A_2|E_1)Pr(A1)$$

Applicazione: verifica moltiplicazione tra matrici

Consideriamo ora un'altra applicazione della randomizzazione all'interno degli algoritmi. Supponiamo di avere 3 matrici nxn, $A,\ B$ e C con elementi in $\{0,1\}$. Vogliamo verificare tale uguaglianza: $A * B = C$.
Un algoritmo banale è quello di moltiplicare A e B e verifare poi con C. Tale algoritmo richiede tempo $\theta(n^3)$, ridotto a $\theta(n^{2.37})$ usando algoritmi sofisticati. Vogliamo adesso usare un algoritmo randomizzato che ci permette di velocizzare tale operazione.

1. Scegliere un vettore $\vec{r} = (r_1, r_2, \ldots, r_n) \in \{0,1\}^n$;
2. Calcolare $B\vec{r}$;
3. Calcolare $A(B\vec{r})$;
4. Calcolare $C\vec{r}$;
5. if $A(B\vec{r}) \neq C\vec{r}$ return $A * B \neq C$ else return $A * B = C$.

Quest'algoritmo impiega $\theta(n^2)$

Come nell'identità polinomiale, questo algoritmo può sbagliare solo in un caso. Ovviamente se $A * B = C$, l' algoritmo non sbaglia mai. Quindi nel caso in cui l'istanza è SI, la probabilità di errore è pari a 0, dunque siamo di frone ad un algoritmo **one-sided-error**. La situazione in cui potrebbe sbagliare è nel caso in cui $A * B \neq C$, ma $AB\vec{r} = C\vec{r}$, ovvero la proiezione della matrice $A * B$ rispetto alla direzione data da $\vec{r}$ va a finire sulla proiezione della matrice C sulla stessa direzione.

Success

Teorema: Se $AB \neq C$ e se il vettore $\vec{r}$ è scelto uniformemente random in $\{0,1\}^n$, allora

$$Pr(AB\vec{r} = C\vec{r}) \leq \frac{1}{2}$$

Info

Lemma: Scegliere $\vec{r} = (r_1, \ldots, r_n) \in \{0,1\}^n$ uniformemente random è equivalente a scegliere ciascun r_i indipendentemente e uniformemente in $\{0,1\}$.
Dunque, la probabilità di scegliere $r_i \in \{0,1\}$ è pari ad $\frac{1}{2}$, quindi scegliere un vettore $\vec{r}$ di dimensione n ha probabilità pari a $\frac{1}{2^n}$.

Computazionalmente, questo si fa andando a scegliere indipendentemente il bit r_i per ciascuna posizione del vettore in modo uniformemente random.

Quando i bit sono scelti in modo indipendente, possiamo usare il **Principle of deferred decision**, che afferma che, anziché prendere tutte le decisioni casuali subito, possiamo prenderle solo quando è necessario farlo. Questo ci permette di mantenere flessibilità durante l'esecuzione e di evitare di prendere decisioni superflue.

Supponiamo di avere una sequenza di bit $r_0, r_1, \ldots, r_t$ e di voler valutare la probabilità che lo XOR dei primi $t-1$ bit sia uguale al $t-esimo$ bit.

$$r_0 \oplus r_1 \oplus r_2 \oplus \cdots \oplus r_{t-1} = r_t$$

Quando stiamo scegliendo il $t-esimo$ bit, i primo $t-1$ sono stati già scelti per il **Principle of deferred decision**, quindi lo XOR dei primo $t-1$ bit assum un valore fissato (deterministico). Da questo momento in poi possiamo analizzare le probabilità in base al valore di r_{t-1}.

- se r_{t-1} vale 0, allora la probabilità che r_t valga 0 è un $\frac{1}{2}$
- se r_{t-1} vale 1, allora la probabilità che r_t valga 1 è un $\frac{1}{2}$

In entrambi i casi (che lo XOR dei primi $t-1$ bit sia 0 o 1), la probabilità che r_t sia uguale o diverso da quello XOR è sempre $\frac{1}{2}$.

Dimostriamo ora il teorema precendente:

Sia $D = AB - C \neq 0$, essendo diversi, la differenza non è nulla. Allora $AB\vec{r} = C\vec{r}$ implica che $D\vec{r} = 0$. Siccome $D \neq 0$ allora ci deve essere un qualche valore diverso da 0 all'interno della matrice, assumiamo senza perdità di generalità che tale valore sia d_{11}.
Per $D\vec{r} = 0$,

$$\sum_{j=1}^{n} d_{1j} r_j$$

oppure sapendo che $d_{11} \neq 0$

$$r_1 = -\frac{\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}$$

Sapendo che gli $r_1, r_2, \ldots, r_n$ sono tutti bit scelti in modo indipendenti con il **Principle of deferred decision** possiamo assumere che gli $r_2, \ldots, r_n$ sono fissati e rimane da scegliere in modo random

r_1.

- se $\frac{\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}$ vale 0 allora la probabilità che r_{11} valga 0 è $\frac{1}{2}$
- se $\frac{\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}$ vale 1 allora la probabilità che r_{11} valga 0 è 0
- se $\frac{\sum_{j=2}^{n} d_{1j} r_j}{d_{11}} \neq$ da 0 e 1 allora la probabilità che r_{11} valga 0 è 0

Quindi in tutti i casi

$$Pr(r_1 = -\frac{\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}) \leq \frac{1}{2}$$

Per migliorare la probabilità di errore, esegue l'algoritmo k volte sulla stessa istanza, ottenendo un probabilità $\leq \frac{1}{2^k}$

Alcune nozioni di probabilità (3)

Success

Teorema (Legge delle Probabilità Totali): Siano $E_1, E_2, \ldots, E_n$ eventi mutualmente disgiunti in Ω tali che $\bigcup_{i=1}^{n} E_i = \Omega$, allora

$$Pr(B) = \sum_{i=1}^{n} Pr(B \cap E_i) = \sum_{i=1}^{n} Pr(B|E_i) Pr(E_i)$$

.

Success

Teorema (Legge di Bayes): Assumiamo che $E_1, E_2, \ldots, E_n$ eventi mutualmente disgiunti tali che $\bigcup_{i=1}^{n} E_i = E$, allora

$$Pr(E_j|B) = \frac{Pr(E_j \cap B)}{Pr(B)} = \frac{Pr(B|E_j) Pr(E_j)}{\sum_{i=1}^{n} Pr(B|E_i) Pr(E_i)}$$

Applicazione: Min-Cut Algorithm

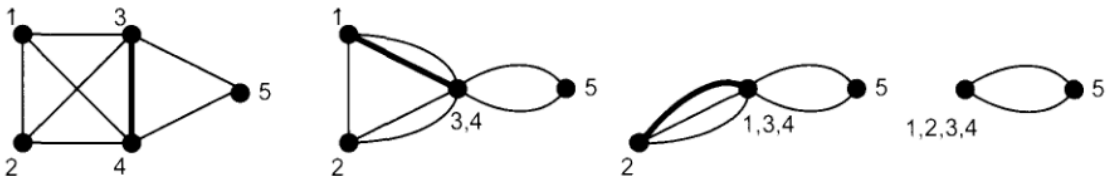
> **Note**
>
> Un **Cut-Set** in un grafo è un insieme di archi la cui rimozione spezza il grafo in due o più componenti connesse. Dato un grafo $G = (V, E)$ con n nodi, il problema del minimo taglio (min-cut) consisite nel trovare il cut-set di cardinalità minima in G.

- **Edge Contraction (Contrazione di un arco)**: È un operazione che permette di semplificare un grafo riducendo il numero di vertici e archi. Quando contrai un arco (u, v), unisci i due vertivi u e v in un unico vertice e gestisci gli archi adiacenti in modo che il grafo risultante mantenga le connessioni rilevanti. Ad ogni contrazione i self loop vengono eliminati.

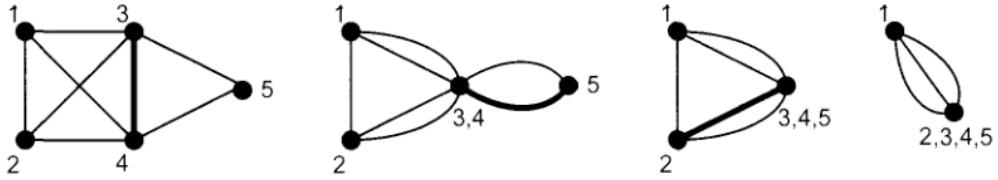
1. Ripeti per $n - 2$ volte:
    i. Segli un arco uniformemente random
    ii. Contrai i 2 nodi connessi da quel arco ed elimina tutti gli archi che collegano i due nodi
2. Ritorna l'insieme di archi che collegano i due 2 nodi rimanenti.

> **Note**
>
> **Invariante**: Ad ogni iterazione, il numero di nodi decresce di 1, dopo $n - 2$ passi, otteniamo sempre un grafo/multigrafo con 2 nodi.

L'algoritmo ogni volta che fa un contrazione di un arco, si deve ricordare i nodi che sono confluiti nel metanodo. Perché poi alla fine dell'algoritmo bisogna ritornare l'insieme degli archi contratti, per cui ha bisogno di ricordare ogni volta qualche arco ha contratto.
Quindi si deve ricordare tutte le contrazzione effettuate nel corso dell'algoritmo.



(a) A successful run of min-cut.



(b) An unsuccessful run of min-cut.

> **Success**
>
> **Teorema**: L'algoritmo restituisce un taglio minimo con probabilità $\geq \frac{1}{n(n-1)}$. Questa rappresenta la probabilità di successo che decresce con l'aumentare di n, però la possiamo abbassare andando sempre a ripetere l'algoritmo k volte sulla stessa istanza.

Se ripeti l'algoritmo indipendentemente per un numero polinomiale di volte e adotti una strategia di selezione (come prendere il minimo o il massimo risultato, a seconda del contesto), puoi ridurre significativamente la probabilità di fallimento.
Se la probabilità di successo in un singolo tentativo è $\frac{1}{p(n)}$, ripetendo l'algoritmo k volte, la probabilità complessiva di fallimento diminuisce esponenzialmente rispetto a k. Con un numero sufficiente di tentativi (dell'ordine dell'inverso della probabilità di successo, eventualmente moltiplicato per un fattore logaritmico per ottimizzare), puoi garantire che almeno uno dei tentativi avrà successo con alta probabilità.

> **Note**
>
> **Lemma**: La contrazione dei archi non riduce la dimensione del minimo taglio. La contrazione può soltanto aumentarla.

Adesso procediamo con l'analisi dell'algoritmo.
Supponiamo di avere un min-cut C di dimensione k. L'obiettivo è quello di stimare la probabilità che nessun arco appartenente a C venga contratto nel processo di contrazione, poiché se un arco del min-cut C viene contratto, l'algoritmo ha *fallito* nel trovare il min-cut corretto.

Andiamo a definire ora i seguenti eventi:

- E_i: L'evento in cui l'arco contratto nell'iterazione i non è in C
- $F_i = \bigcap_{j=1}^{i} E_i$: L'evento in cui nessun arco di C è stato contratto nelle prime i iterazioni.

Dobbiamo quindi calcolare la probabilità $Pr(F_{n-2})$. Iniziamo calcolando $Pr(E_1) = Pr(F_1)$. Poiché il min-cut ha dimensione k, tutti i nodi del grafo devono avere grado almeno k, allora il numero di archi è almeno $\frac{nk}{2}$. Il primo arco è scelto uniformemente random dal insieme degli archi, dunque

$$Pr(E_1) = Pr(F_1) \geq 1 - \frac{k}{\frac{nk}{2}} = 1 - \frac{2}{n}$$

Supponiamo ora che la prima contrazione non abbia eliminato un arco di C, quindi andiamo a condizionare su F_1. Ora abbiamo un grafo con cut-set di dimensione sempre k (appunto perché abbiamo supposto di non aver eliminato un arco di C) e avente $n-1$ nodi e di conseguenza $\frac{(n-1)k}{2}$ archi, dunque

$$Pr(E_2|F_1) \geq 1 - \frac{k}{\frac{(n-1)k}{2}} = 1 - \frac{2}{n-1}$$

Generalizzando, otteniamo

$$Pr(E_i|F_{i-1}) \geq \frac{k}{\frac{(n-i+1)k}{2}} = 1 - \frac{2}{n-i+1}$$

Per calcolare infine $Pr(F_{n-2})$ usiamo $Pr(A \cap B) = Pr(A|B)Pr(B)$, e andando a iterare otteniamo che

$$Pr(F_{n-2}) = \frac{2}{n(n-1)}$$

Siccome l'algoritmo è un **one-sided-error**, possiamo ridurre la probabilità ripetendo l'algoritmo per $n(n-1)ln(n)$ volte e andando poi a sciegliere come minimo taglio, in minimo di tutte le iterazioni, allora la probabilità che l'output non è il minimo taglio è delimitata da

$$\left(1 - \frac{2}{n(n-1)}\right)^{n(n-1)ln(n)} \leq e^{-2ln(n)} = \frac{1}{n^2}$$

> **Note**
>
> Nella disuguaglianza sopra è stata usata la regola $1 - x \leq e^{-x}$

Applicazione: Quick Sort

Il Quick Sort è un algoritmo di ordinamento. Dato un insieme S di elementi, li ordina in modo crescente o decrescente. Tale algoritmo
ha complessità worst case $O(n^2)$. Osserviamo ora 2 versione del Quick Sort che migliorano la complessità worst case.

RQS

Note

Procedure Random-Quick-Sort(S)
1. Scegli uniformemente random un elemento y di S
2. Confronta gli elementi di S con y e siano

$$S_1 = \{x \in S - \{y\} | x \leq y\},\ S_2 = \{x \in S - \{y\} | x > y\}$$

3. Return Random-Quick-Sort(S_1),y,Random-Quick-Sort(S_2)

Dunque abbiamo modificato il Quick-Sort in modo da scegliere il pivot in modo randomico. La radomizzazione fa sì che è molto improbabile che vengano scelti ripetutamente i perni sbagliati. Dimostreremo in seguito che il numero di confronti che esegue il RQS è $2nln(n) + (n)$ rientrando dunque nel lower bound $\Omega(nln(n))$ degli algoritmi di ordinamento basati su confronto.

Success

Teorema: Supponiamo che, quando avviene la scelta random del pivot nel RQS, esso è scelto indipendentemente dagli altri e uniformemente random. Allora, per ogni input, il numero atteso di confronti eseguiti dal RQS è $2nln(n) + O(n)$.

Dimostrazione: Siano $y_1, y_2, \ldots, y_n$ gli elementi di S ordinati in ordine crescente. Per $i < j$, definiamo X_{ij} una variabile aleatoria che assume valore 1 se due elementi y_i e y_j sono stati confrontati durante l'esecuzione dell'algortimo, altrimenti 0. Allora il numero totale di confronti è

$$X = \sum_{i=1}^{n} \sum_{j>i} X_{ij}$$

Noi siamo interessati dunque a calcolare $E[X] = E\left[\sum_{i=1}^{n}\sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n}\sum_{j>i} E[X_{ij}]$.
Dobbiamo adesso calcolare quanto vale $Pr(X_{ij}) = 1$. y_i è confrontato con y_j se soltanto se uno dei due è scelto come pivot dall'insieme $Y^{ij} = \{y_i, y_{i+1}, \ldots, y_{j-1}, y_j\}$. Questo perche se y_i $(o\ y_j)$ è il primo pivot selezionato da questo insieme, allora y_i e y_j dovranno stare nella stessa sottolista, e di conseguenza verranno confrontati una volta sola. Altrimenti, se nessuno dei due è stato scelto come pivot, y_i e y_j saranno separati in due sottoliste distinte e non verranno mai confrontati. Siccome i pivot sono scelti uniformemente random dall'insieme Y^{ij}, allora la probabilità che essi verranno scelti è: $Pr(X_{ij} = 1) = \frac{2}{j-i+1}$. Andando a fare i calcoli otteniamo che

$$E[X] \leq n \sum_{k=1}^{n} \frac{2}{k} \leq 2nH_n = nln(n) + O(n)$$

DQS

Note

Procedure Deterministic-Quick-Sort(S)
1. Sia y il primo elemento di S
2. Confronta gli elementi di S con y e siano

$$S_1 = \{x \in S - \{y\} | x \leq y\},\ S_2 = \{x \in S - \{y\} | x > y\}$$

3. Return Deterministic-Quick-Sort(S_1),y,Deterministic-Quick-Sort(S_2)

Dunque in questo caso usiamo l'algoritmo deterministico del Quick Sort ma consideriamo un modello probabilistico per l'input. Una *permutazione* di un insieme di n elementi è solo una delle $n!$ possibili ordinamenti degli elementi di quest'insieme. Invece di trovale il peggior input possibile, assumiamo che l'input ci è stato dato in modo random.

Success

Teorema: Supponiamo che, quando avviene la scelta random del pivot nel RQS, esso è il primo elemento della sottolista. Se l'input è scelto uniformememente random da tutte le sue permutazioni di valori, allora, il numero atteso di confronti eseguiti dal RQS è $2n\ ln(n) + O(n)$.

Dimostrazione: La dimostrazione è essenzialmente simile a quella del RQS. Come prima, y_i e y_j sono confrontati se soltanto se uno dei due è scelto come pivot dall insieme Y^{ij}. Siccome l'ordine degli elementi in ciascuna sottolista è lo stesso come nell'originale, allora il primo pivot selezionato dall'insieme Y^{ij} è il primo elemento dell'insieme e poiché tutte le possibili permutazioni dei valori di input sono ugualmente probabili, ogni elemento in Y^{ij} è ugualmente probabile per essere il primo elemento. Da qui possiamo usare la linearità della speranza matematica e otteniamo che

$$E[X] = E\left[\sum_{i=1}^{n}\sum_{j>i} X_{ij}\right] = O(n\ ln(n))$$

Applicazione: Calcolo del mediano

Sia S un insieme di n di elementi di un universo totalmente ordinato, il **mediano** di S è l'elemento m tale che se S è ordinato
allora m divide a metà S, ovvero $\frac{n}{2}$ elementi sono più piccolo di m e $\frac{n}{2} + 1$ elementi sono più grandi di m.

$$S = \{75, 10, 29, 48, 43, 12, 6\}$$

$$S_{ord} = \{6, 10, 12, 29, 43, 48, 75\}$$

Allora il mediano è il numero 29.

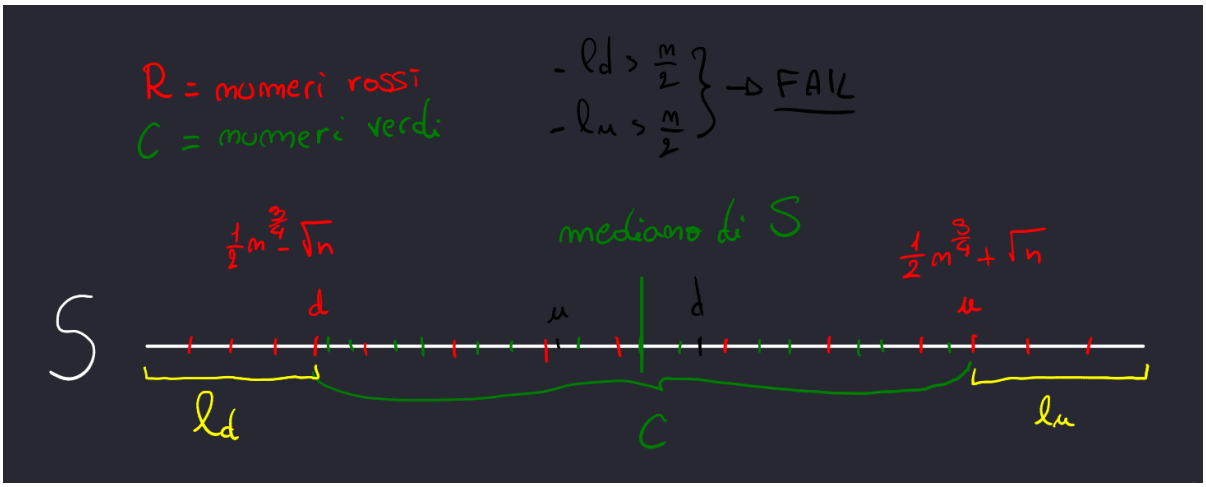
Il mediano può essere computato deterministicamente in tempo $O(nlog(n))$ ordinando S oppure con un algoritmo molto sofisticato e complesso in tempo $O(n)$. Andiamo a adesso a vedere un algoritmo randomizzato per il calcolo del mediano che ha complessità $O(n)$, dovendo però per semplicità assumere che n è dispari e gli elementi di S sono tutti distinti.

Note

Procedure Compute-Median(S)

1. Prendi un multi-set R di $s = n^{\frac{3}{4}}$ elementi di S scelti indipendentemente e uniformemente random con ripetizione, ovvero un elemento può essere scelto più di una volta. Ordina l'insieme R.
2. Sia d il $(\frac{1}{2}n^{\frac{3}{4}} - \sqrt{n})$ - esimo elemento più piccolo nell'insieme ordinato R.
3. Sia u il $(\frac{1}{2}n^{\frac{3}{4}} + \sqrt{n})$ - esimo elemento più piccolo nell'insieme ordinato R.
4. Confrontando ciascun elemento di S con d e u, calcola l'insieme $C = \{x \in S : d \leq x \leq u\}$, e i numero $l_d = \left| \{x \in S : x < d\} \right|$ e $l_u = \left| \{x \in S : x > u\} \right|$.
5. Se $l_d > \frac{n}{2}$ oppure $l_u > \frac{n}{2}$ allora **FAIL**.
6. Se $\left| C \right| \leq 4n^{\frac{3}{4}}$ allora ordina C, altrimenti **FAIL**.
7. Ritorno il primo $(\frac{n}{2} - l_d + 1)$ elemento dell'insieme C ordinato.

L'idea dell'algoritmo consiste nel *sampling (campionamento)*.



- d è il mediano di R shiftato a sinistra di $\sqrt{(n)}$.
- u è il mediano di R shiftato a destra di $\sqrt{(n)}$.
- Calcolo C in tempo $O(n)$ con tutti gli elementi di S che sono compresi tra u e d, quindi tra i due mediani shiftati.
- C però non inizia dal primo elemento di S, ma è shiftato di quanti elementi ci sono prima di d, e sono l_d.
- Cosa significa che $l_d > \frac{n}{2}$? Significa che d si trova oltre il mediano. Stessa cosa se $l_u > \frac{n}{2}$. In questo caso il campione è sbilanciato o verso destra o verso sinistra e allora **FAIL**, quindi il mediano non è centrale ma spostato a destra/sinistra di un fattore $\sqrt{(n)}$. Dobbiamo dimostrare che questi 2 eventi accadono con una bassa probabilità.
- Un altro evento cattivo è se C è molto più grande di $O(n^{\frac{3}{4}})$. Ordiniamo C perché essendo un ibrido di S e R e poiche S non è ordinato, dobbiamo per forza ordinarlo, ma per ordinarlo in poco tempo, la grandezza di C deve essere $\leq 4n^{\frac{3}{4}}$.

Gli elementi di R vengono scelti con probabilità $\frac{1}{n^{\frac{1}{4}}}$, la domanda che ci viene spontanea è, perché $n^{1/4}$? Tutto nasce dalla *Chebyshev inequality* e vogliamo qulcosa che abbia una sicurezza data dalla deviazione standard.
R a sto punto diventa un insieme random di dimensione $n\frac{1}{n^{1/4}} = n^{3/4}$. Quindi per esempio, se $n = 10^{10}$, la dimensione di $R = (10^{10})^{3/4}$. La nostra speranza è che tutti gli elementi di R siano ben spaziati su tutto l'insieme S.

Ordinare R (di size $n^{\frac{3}{4}}$) con un metodo standard quindi con un algoritmo che richiede $O(m\ log(m))$ ci costa $\frac{3}{4}\ n^{\frac{3}{4}} log(n) = o(n)$
In un caso ottimale, il mediano di R corrisponde con il mediano di S, ma sappiamo tutti che c'è una probabilità di errore. Infatti il mediano si può trovare o più a *destra* o più a *sinistra*, quindi il mediano può essere decentrato ma di quanto? Di $-\sqrt{(}n)$ o $\sqrt{(}n)$.

Andiamo adesso ad analizzare quest'algoritmo. Abbiamo dunque visto che l'algoritmo fallisce nel computare il mediano se succede uno dei 3 eventi E_1, E_2, E_3.

- Sia Y_1 il numero si campioni sotto il mediano in R. Dunque $Y_1 = \sum_{i=1}^{n^{\frac{3}{4}}} Y_1^i$ dove $Y_1^i = 1$ se soltanto se $x_i < \ mediano$, altrimenti vale 0, quindi Y_1^i è una variabile aleatoria binaria.
- Sia Y_2 il numero di campioni sopra il mediano in R.

Defiano ora gli eventi:

- $E_1 : \ Y_1 < \frac{1}{2}n^{\frac{3}{4}} - \sqrt{(}n)$, ovvero R è spostato troppo a destra, ovvero d è sposato troppo a destra.
- $E_2 : \ Y_2 < \frac{1}{2}n^{\frac{3}{4}} - \sqrt{(}n)$, ovvero R è spostato troppo a sinistra, ovvero u è sposato troppo a sinistra.
- $E_3 : \left| \ C \ \right| > \frac{n}{log(n)}$, ovvero R non è ben distribuito.

Iniziano con calcolare la probabilità che accada l'evento E_1.

$Y_1 \sim Bernoulli(p = \frac{1}{2})$, vedendo Y_1 come la somma di $n^{\frac{3}{4}}$ variabili Bernoulliane indipendenti con

$$E[Y_1] = \sum_{i=1}^{n^{\frac{3}{4}}} E[Y_1^i] = \sum_{i=1}^{n^{\frac{3}{4}}} \frac{1}{2} * 1 + \frac{1}{2} * 0 = \frac{1}{2}n^{\frac{3}{4}}$$

$$Var[Y_1] = \sum_{i=1}^{n^{\frac{3}{4}}} Var[Y_1^i] = \frac{1}{4}n^{\frac{3}{4}}$$

Adesso usando la **Chebyshev inequality**

$$Pr(\left| X - E[X] \right| > a) \leq \frac{Var[X]}{a^2}$$

Otteniamo che:

$$Pr(E_1 : Y_1 < \frac{1}{2}n^{\frac{3}{4}}) \leq Pr(\left| Y_1 - E[Y_1] \right| > \sqrt{(}n)) \leq \frac{Var[Y_1]}{n} = \frac{1}{4}n^{\frac{-1}{4}}$$

Similmente:

$$Pr(E_2 : Y_2 < \frac{1}{2}n^{\frac{3}{4}}) \leq Pr(\left| Y_2 - E[Y_2] \right| > \sqrt{(}n)) \leq \frac{Var[Y_2]}{n} = \frac{1}{4}n^{\frac{-1}{4}}$$

Infine:

$$Pr(E_1 \cup E_2) = \frac{2}{4}n^{\frac{-1}{4}}$$

Quindi questa probabilità è piccola e tende a 0 come l'inverso di un polinomio, quindi c'è alta probabilità che questo evento non succeda.

$Pr(E_3)$ da fare quando me va, forse mai.

Applicazione: Content Resolution

Supponiamo di avere n processi, $P_1, P_1, \ldots, P_n$ ciascuno in competizione per accedere ad una risorsa condivisa, per esempio un database. L'accesso alla risorsa deve essere in mutua esclusione, ovvero se due o più processi provano ad accedere alla risorsa in contemporanea, allora tutti i processi vengono bloccati.

I processi in un sistema distribuito non possono comunicare direttamente tra loro, e non è possibile uno scheduling deterministico, come ad esempio uno schema in cui i processi siano numerati da 1 a n e al tempo i acceda il processo i (un tipo di Round Robin). Questo approccio richiederebbe che i processi fossero etichettati in modo fisso, ma in un sistema distribuito tale etichettatura non è attuabile. Inoltre, l'insieme dei processi potrebbe non essere statico: capita spesso che un processo sia inattivo in un certo momento o che se ne aggiungano di nuovi. Di conseguenza, uno scheduling deterministico risulta impraticabile.

Quindi uno schema tipico che si usa per lo scheduling nei sistemi distribuiti è la randomizzazione.

Come funziona l'algoritmo:
Qui c'è un problema classico che nei sistemi distribuiti viene chiamato **symmetry breaking**, è una

tecnica usata nei sistemi distribuiti per prevenire che due processi eseguono le stesse azioni in simultaneo.

Ciascun processo ad ogni round (quindi ad ogni istante di tempo) richiede l'accesso al database/risorsa con probabilità $p = \frac{1}{n}$ e con probabilità $1 - p$ di non accedere.
Il round ha successo se esattamente un processo accede alla risorsa e tutti gli altri non accedono.

Inziamo con definire alcuni eventi.

- Sia $S[i, t]$: l'evento che P_i tenta di accedere al database al tempo t. Usando $p = \frac{1}{n}$ possiamo massimizzare questa probabilità ottendo che

$$Pr(S[i, t]) = p(1 - p)^{n-1} = \frac{1}{n}(1 - \frac{1}{n})^{n-1}$$

Perchè la probabilità è $p(1 - p)^{n-1}$? Il processo P_i chiede di accedere alla risorsa con probabilità p mentre i restanti $n - 1$ processi non devono acccedere alla risorsa con probabilità $1 - p$.

Quindi la probabilità che un round abbia successo è dell'ordine di $\frac{1}{n \cdot e}$ in quanto $\left(1 - \frac{1}{n}\right)^{n-1}$ per n molto grande tende a $\frac{1}{e}$

TODO: Da finire.