ASD - Algoritmi e Strutture Dati

Elia Ronchetti
@ulerich

2022/2023

Indice

Capitolo 1

Introduzione algoritmi

1.1 Che cos'è un algoritmo?

Un algoritmo è

- Una sequenza di istruzioni elementari
- Agisce su un input per produrre un output
- Risolve un problema computazionale

Un algoritmo deve essere corretto e efficiente.

Corretto Significa che deve funzionare per qualsiasi input valido

Efficiente Deve occupare il minor spazio possibile ed impiegare il minor tempo possibile.

L'efficienza di un algoritmo si misura in termini di spazio e tempo

1.2 Analisi di un algoritmo

Per analizzare l'efficienza di un algoritmo si calcola il numero di istruzioni eseguite, ma esso non è univoco, varia in base all'input ricevuto, è quindi necessario individuare il **caso migliore** e il **caso peggiore**, essi si analizzano a parità di dimensioni, per questo non dipendono da essa. Dire che il caso migliore è quando l'array è vuoto non ha senso ai fini dell'analisi.

Per avere un'idea dei tempi di esecuzione è necessario calcolare il **Caso Medio**

NON è la media tra caso peggiore e caso migliore!

1.3 Regole sullo Pseudocodice

Gli algoritmi saranno scritti in Pseudocodice secondo le seguenti regole

- Il codice sarà simil C/Java
- Cicli: for, while, do-while
- Condizioni: if, else
- Indentazione + begin/end
- Commenti /*.....*/
- Assegnamenti A = 5, A := 5, A $\leftarrow$ 5
- Test del valore A == 5
- Variabili: locali
- Array A[i] $\rightarrow$ i $\rightarrow$ 1 ... n
- Gli Array partono da 1
- Dati sono considerati oggetti con attrivuti (come length(A) per gli array)
- Puntatori: liste dinamiche
- Funzioni/Procedure - I parametri sono passati per valore (non per indirizzo)

Macchina RAM (Random Access Machine) La macchina su cui verranno eseguiti gli algoritmi sarà considerata RAM e quindi con le seguenti Caratteristiche

- Memoria ad accesso diretto
- No limiti memoria
- Sistema monoprocessore

1.4 Esame

L'esame sarà uno scritto con esercizi e domande di teoria. I parziali sono tendenzialmente riservati al primo anno, ma è possibile scrivere una email al prof 2 settimane prima del parziale e chiedere di poterlo sostenere anche se si è di un altro anno, sarà a sua discrezione concedere o meno questa opportunità. Si possono recuperare i parziali, è possibile anche tentare un recupero per migliorare un voto già positivo, accettando il rischio di che se il voto preso nell'esame di recupero è minore di quello originale si dovrà accettare quel voto.

1.5 Definizioni di base

Algoritmo Corretto Un algoritmo si definisce corretto se per tutti gli input si ottiene il risultato desiderato, l'algoritmo è corretto solo se garantisce la correttezza del risultato.

Algoritmo efficiente Minor utilizzo di **Spazio** e **Tempo**.

Determinare l'efficienza di un algoritmo

Il primo passo è determinare il numero di istruzioni eseguite dall'algoritmo dato che così facendo non dipendo dalla potenza dell'hardware e dall'input.

Operazioni valutazione algoritmo

1. Conto le istruzioni **eseguite**

2. Determinare T peggiore - T migliore - T medio (la media non è fra T peggiore e T migliore)

Il tempo non sarà una quantità in secondi, ma dipenderanno da un parametro n T(n).
Quando devo scegliere un algoritmo mi baso sulla funzione n, dato che al crescere dell'input la funzione crescerà in modo lineare, quadratico, cubico, ecc. e questo mi mostrerà come cresce il tempo in funzione di n.
A parità di n controllo il fattore moltiplicativo.

Esempio I polinimoniali hanno tempi di esecuzione accettabili, mentre i tempi esponenziali sono intrattabili, il problema è che esistono algoritmi esatti, ma sono esponenziali, per questo sono inutili dato che non con grandi input non si fermano mai.
Determinare il **Caso peggiore** serve per capire quanto tempo devo aspettare al massimo, quindi dopo quanto tempo avrò sicuramente un risultato, il **Caso minore**, determina il tempo minimo che devo aspettare, il **Caso medio** determina mediamente quanto tempo devo aspettare (non è la media fra T peggiore e T migliore).

1.6 Ricerca Sequenziale

```
int RicSeq(int k, int v[])
 1  i=1
    while v[i] != k and i <= length(v)
        i++
 1  if i <= length(v)
1/0     return(i)
    else
1/0     return(-1)
```

In questo semplice algoritmo per la ricerca sequenziale di un numero all'interno di un array ci concentreremo sulla ricerca del caso peggiore e quello migliore.
Ricordiamo che il caso peggiore e migliore si determina a parità di dimensione dell'input, non ha senso dire che il caso migliore è il vettore vuoto.
I numeri a fianco indicano il numero di volte per cui ogni operazione è eseguita. Else non viene considerato dato che non viene effettuato un controllo, quando l'if è falso rimanda alle istruzioni sotto l'else (a livello di linguaggio macchina è un'etichetta che indica al codice dove effettuare la jump nel caso in cui la condizione dell'if non fosse vera).

1.6.1 Cosa devo identificare

Devo identificare:

- Quando si verifica (in che condizioni)
- Il tempo di esecuzione

Quando si verifica

Migliore Indicato come **t**. Posso pensare (sempre intuitivamente) che il caso migliore è quando il numero si trova nella prima posizione del vettore. Effettivamente in questo caso il ciclo while non viene mai attuato (a parte la verifica della condizione)

Peggiore In questo caso devo identificare per quale input la ricerca sequenziale performa peggio, intuitivamente posso pensare che il caso peggiore è quando il numero non è presente nel vettore, dato che devo scorrere tutto il vettore.
Analizzando l'esecuzione di tutte le istruzioni posso confermare la mia ipotesi.

Importante Non basta fare un esempio, per identificare il caso migliore e peggiore, devo descrivere (anche in forma verbale) per quali input si verifica, in questo caso abbiamo detto infatti, quando il numero non è presente oppure quando il numero si trova nella prima posizione.

Analisi tempo di esecuzione Il tempo di esecuzione dipende dal numero di operazioni eseguite, i tempi di esecuzioni maggiori si concentrano spesso nei cicli (for, while, do-while, ecc.), ma è comunque importante valutare tutte le istruzioni.

Tempo di esecuzione caso migliore Indicato con $t(n)$ In questo caso vengono eseguite solamente 4 istruzioni

```
int RicSeq(int k, int v[])
 1   i=1
 1   while v[i] != k and i <= length(v)
           i++
 1   if i <= length(v)
 1         return(i)
     else
 0         return(-1)
```

Le quattro istruzioni con dei commenti

- i=1
- while (controlla una volta sola dato che trova subito che k = v[i])
- if i $\leq$ length(v)
- return(i)

Si tratta di una funzione costante $t(n) = 4$, solitamente le funzioni riguardanti i tempi di esecuzione sono in funzione di n, ma in questo specifico caso la funzione non dipende dalla dimensione dell'input, infatti se il numero è all'inizio del vettore a prescindere dalle dimensioni dell'array non dovrò mai scorrere il vettore (vedremo che non è praticamente mai così per il caso peggiore)

Analisi esecuzione caso peggiore Indicato con $T(n)$

```
int RicSeq(int k, int v[])
 1   i=1
n+1  while v[i] != k and i <= length(v)
 n       i++
 1   if i <= length(v)
 0       return(i)
   else
 1       return(-1)
```

Il ciclo while viene eseguito n+1 volte dato che ogni volta che incremento i++, devo verificare di non essere uscito dall'array, quindi verrà eseguito sempre una volta in più rispetto all'incremento.
Questo significa che $T(n) = 3 + 1 + 2n = 2n + 4$ dove n è la dimensione dell'input, in questo caso la dimensione dell'array.
Questa funzione indica il numero di istruzioni eseguite (non è propriamente un tempo).

1.6.2 Caso medio

Indicato come $T_m(n)$, è il tempo medio di esecuzione, non è la media fra il caso migliore e peggiore.
In questo caso dobbiamo chiederci, cosa ci aspettiamo che succeda mediamente?
In questo caso mi aspetto che il caso medio sia che k si trovi in posizione $\frac{n}{2}$, quindi che sia a metà. Mi aspetto quindi che il caso medio sia il caso peggiore diviso due.
In generale il caso medio è un po' più complesso dato che richiederebbe di conoscere la distribuzione di probabilità dell'input.
$T_m(n) = 2\frac{n}{2} + 4 = n + 4$

Possibile obiezione Sto mettendo sullo stesso piano le istruzioni if e i while che magari hanno più condizioni da verificare, effettivamente il while impiega leggermente più tempo, posso scrivere una funzione indicando con c $(c_1, c_2, c_3, \dots)$ il tempo di esecuzione che può avere ogni istruzioni, il problema è che la funzione diventa enorme e poco comprensibile, alla fine a me interessa capire l'ordine di grandezza di esecuzione per poter capire se l'algoritmo è efficiente e per poterlo confrontare con altri algoritmi, alla fine se confronto un algoritmo che ha come caso medio $1000n$ con uno che ha come caso medio n^2 sceglierò comunque il primo, quindi non mi interessa se alcune funzioni ci mettono un po' di più di altre.