Programmazione
di
Processori *MultiCore* – I lezione

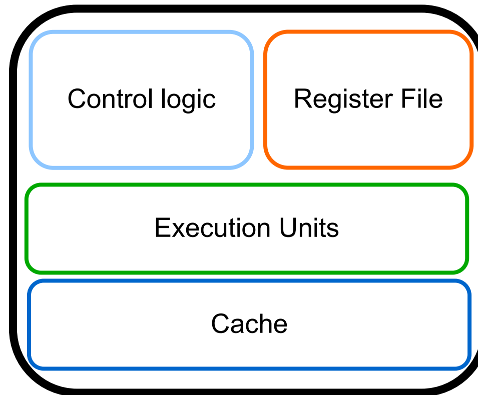
Federico Massaioli (`federico.massaioli@caspur.it`)

CASPUR e Università degli Studi di Roma "La Sapienza"
Laurea Magistrale in Informatica
Anno accademico 2008-2009

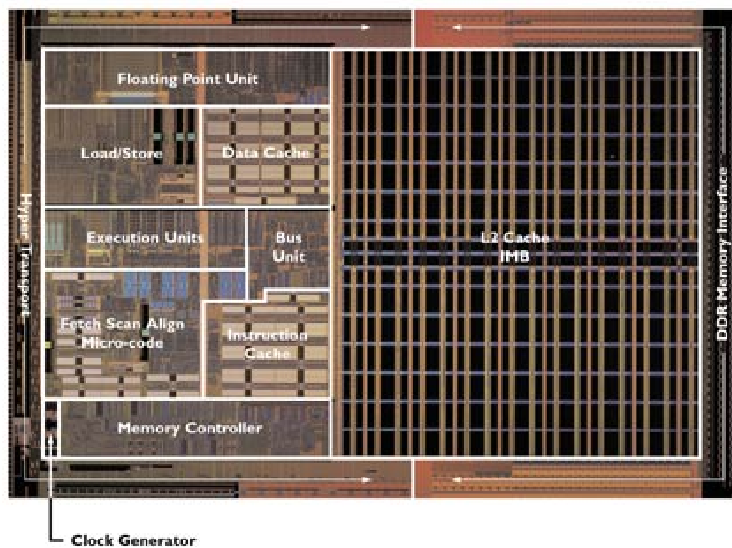
Risorse del corso

- Pagina su twiki.di.uniroma1.it (da aggiornare…)
- *Mailing list*: `multicore@inroma.roma.it`
- Per l'iscrizione: `https://lists.inroma.roma.it/mailman/listinfo/multicore`
- INDICANDO NOME E COGNOME!!!

CPU *Single core*



AMD Opteron



Legge di Moore 1965÷2004

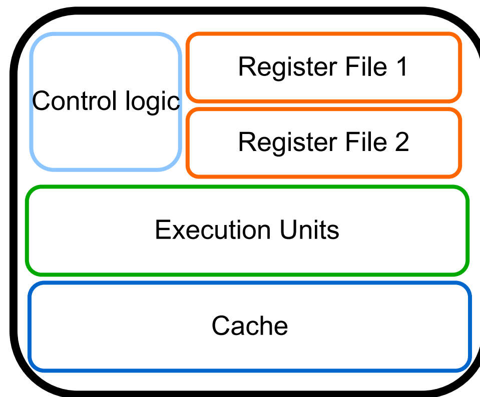
- Il numero di transistor sul *chip* raddoppia ogni 18 mesi
- Raddoppio delle unità funzionali
  - unità di calcolo: 2× capacità di calcolo
  - logica di controllo: ~4× complessità
  - 8× potenza assorbita
- Raddoppio del *clock*
  - unità di calcolo: 2×velocità di picco
  - 8×potenza assorbita
- La reale efficienza dei codici:
  - dipende dall'efficienza dell'applicazione
  - è limitata dalle dipendenze tra operazioni
  - l'ottimizzazione del codice è importante

Performance: teorica e reale

Prestazioni teoriche (di "picco") e "*sustained*" di differenti CPU, per un codice cinetico, in MFlops:

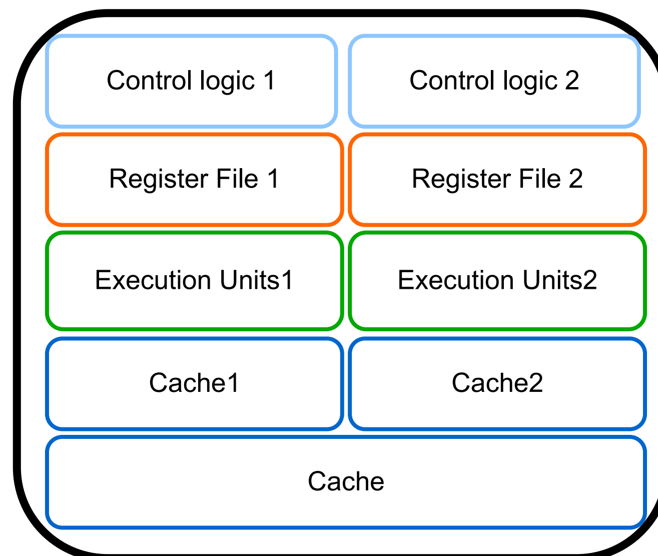
CPU	Picco	*Sustained*	Percentuale
HP EV67@1.25 Ghz	2500	1400	56 %
Opteron@2.0 Ghz	4000	1340	34 %
IBM Power4@1.3 Ghz	5200	889	19 %
Itanium2@1.5 Ghz	6000	2080	35 %
IBM G5@2.0 Ghz	8000	1100	14 %
Nec SX6i@0.5 Ghz	8000	6000	75 %

Single core + SMT

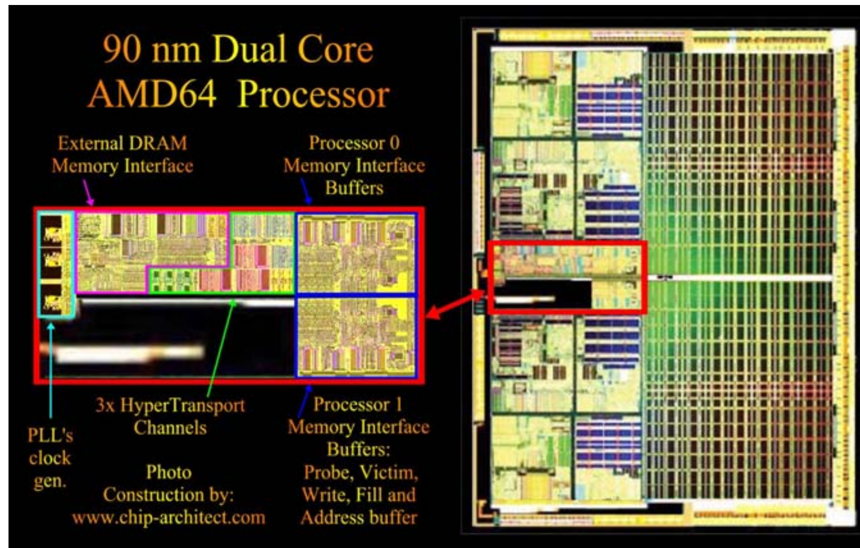


- Più flussi di codice indipendenti per sfruttare la potenza di calcolo
- Logica di controllo molto complicata
- Necessario ottimizzare prima di parallelizzare
- Poco efficace per il calcolo scientifico

Dual core



AMD Opteron *dual core*



Legge di Moore 2005÷…

- Il numero di transistor sul *chip* raddoppia ogni 18 mesi
- *N* CPU *on-chip*
  - 1× velocità di picco
  - *N*× capacità di calcolo
  - *N*× potenza assorbita
- *Clock* costante → velocità di picco costante
- È cruciale ottimizzare l'applicazione
- È cruciale parallelizzare il codice
- È inutile parallelizzare un codice inefficiente

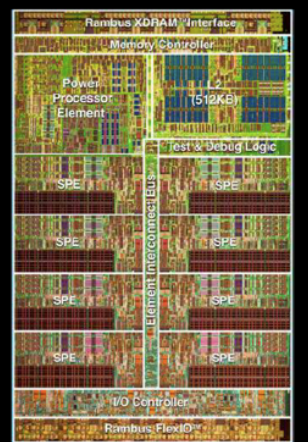
E già si parla di *many-core*…

- Sun Niagara, IBM Cell, Nvidia GPU, Intel Tera-scale, …
- Decine/centinaia di *core* semplificati sullo stesso *chip*
- Logica di controllo semplificata
  - niente riordinamento automatico delle istruzioni
  - il *core* “stalla” quando i dati non arrivano dalla memoria
- L’ordinamento delle operazioni nel codice sarà fondamentale per l’efficienza

IBM Cell (1)

Highlights (3.2 GHz)

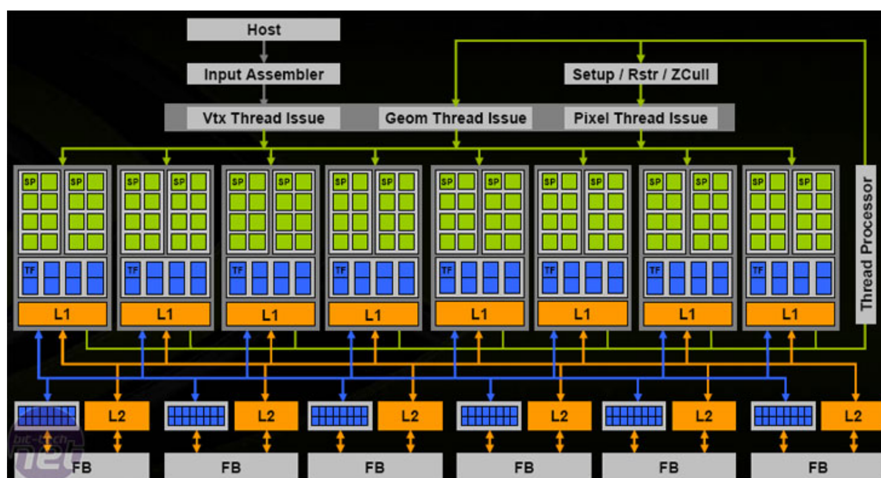
- **241M transistors**
- **235mm2**
- **9 cores, 10 threads**
- **>200 GFlops (SP)**
- **>20 GFlops (DP)**
- **Up to 25 GB/s memory B/W**
- **Up to 75 GB/s I/O B/W**
- **>300 GB/s EIB**
- **Top frequency >4GHz** (observed in lab)



IBM Cell (2)

- Processore che equipaggia al PS3!!
- Nato per il mercato *consumer*
  - videogiochi
  - HDTV
  - *Digital Signal Processing*
- Non progettato per il mercato tecnico/scientifico
- 9 *core*: 1 processore IBM PowerPC
  + 8 *Synergistic Processing Elements* (SPE)
  - SPE: funzionalità ridotte, unità SIMD
  - SPE: una memoria locale di 128 KB
  - SPE: Ottimizzati per interi e singola precisione (204 GFlops), doppia più lenta (25 GFlops)

GPU Nvidia GeForce 8 (1)



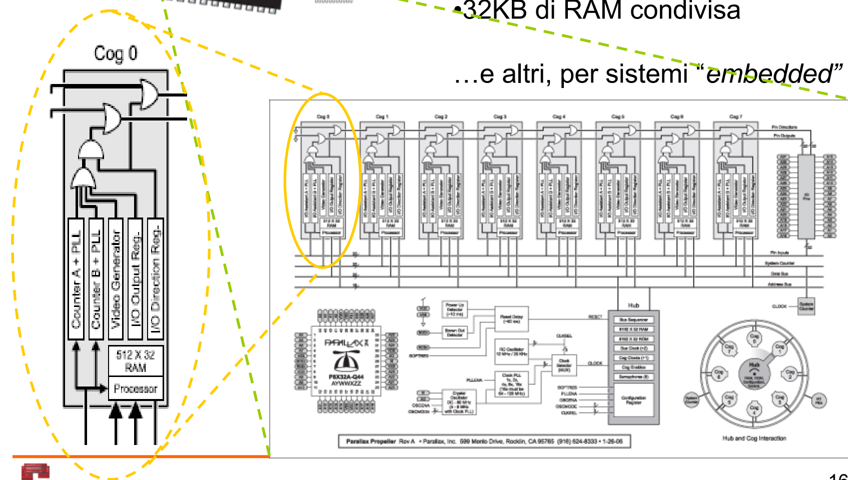
GPU Nvidia GeForce 8 (2)

- Nata per la grafica *real-time*
- Consiste di 128 *stream processors* (SPs)
  - Unità di calcolo seriali capaci di completare
    - una somma+prodotto in singola precisione
    - un prodotto in singola precisione
  - Il *clock* non è uguale dappertutto
    - SP 1.35 Ghz
    - *Chip*: 0.575 Ghz
  - *Performance* (teorica di picco)
    - 128*(2+1)*1.35 = 518 Gflops
  - Uso estensivo del paradigma SIMD
- E' possibile usarla per calcolo numerici (via CUDA)
  - Solo singola precisione (per ora)
  - Implica modifiche pesanti del codice
- Versioni future dedicate al solo calcolo

Non solo grandi *performance*…

Parallax' Propeller:
•8 core, fino a 80MHz
•2KB×8 RAM locale
•32KB di RAM condivisa

…e altri, per sistemi *"embedded"*



Performance: chi sviluppa compilatori

- Basta “scrivere bene” il codice, “organizzare bene” i dati in memoria ed “usare bene” il compilatore
  - prodotto matrice-matrice in Fortran
  - macchina IBM Power4@1100 Mhz
  - matrici 1024×1024, doppia precisione

Opzione	secondi	MFlops
`-O0`	24’’	89
`-O2`	6.35’’	338
`-O3`	4.87’’	487
`-O4`	2.14’’	1003
`-O5`	1.93’’	1111

- Perché tanta varietà di risultati?
- Basta in ogni caso passare da **`-On`** a **`-On+1`**?

Performance: chi realizza sistemi

- Quanta percentuale di CPU è utilizzata?
- Quante CPU?
- Sono usate in modo bilanciato?
- Quanta memoria è utilizzata?
- Il sistema pagina?
- Le comunicazioni sono efficienti?
- Quante chiamate di sistema vengono fatte?
- È efficiente l’*Input/Output*?
- Qual è il *throughput* complessivo?
- Qual è il *throughput* per Watt?

Performance: chi inventa algoritmi

Complessità dell'algoritmo: numero di "operazioni" in funzione della dimensione N del problema

Metodo	Anno	**Complessità**
Eliminazione Gaussiana	1945	N^7
SOR (suboptimal)	1954	$8 N^5$
SOR (optimal)	1960	$9 N^4 Log(N)$
Cycle Reduction	1970	$8 N^3 Log(N)$
Multigrid	1978	$60 N^3$

Attenzione:

- proprietà numeriche differenti?
- costanti a moltiplicare?
- uso di memoria?

Performance: l'utente

- Avere risultati corretti nel minor tempo possibile
- Il cliente ha sempre ragione!

- Tutti i fattori contano:
  - algoritmo
  - implementazione (codice)
  - uso del compilatore
  - I/O ed interazioni con l'OS
- Sono necessari strumenti diversi

Performance: come ottenerle

- Ridurre numero di operazioni → algoritmica
- Velocizzare scambio dati CPU - memoria → implementazione
- Usare operazioni meno onerose → algoritmica "spicciola"
- Massimizzare attività della CPU → implementazione
  - a volte in "contrasto" con i punti precedenti
- Scrivere il codice in modo da agevolare il compilatore
  - nei casi più complessi: ottimizzazioni a mano
- Non reinventare la ruota: usare librerie

Ambiente di esecuzione

- L'applicazione è eseguita in un ambiente complesso
  - *Hardware*: CPU, memoria, dispositivi
  - *Software*: OS, librerie, altre applicazioni in esecuzione
- L'applicazione ha "l'illusione" di una macchina virtuale molto semplice e *monotasking*
- Ma interagisce, direttamente o indirettamente, con tutte le componenti

hello world

```
#include <stdio.h>
main()
{
    int i;
    for(i=1;i<9;++i)
        {
            printf ("Hello World!\n");
        }
}
```

- Interazioni Hardware
  - Device di I/O
  - Bus di Memoria
  - Livelli di Memoria
- Interazioni Software
  - Librerie
  - Runtime

hello world (assembler - 1)

- Assembler su IBM Power5

```
IBM XL C Enterprise Edition for AIX, Version 7.0.0.4 --- hello.c 07/09/06 12:45:09 (C)
>>>>> OPTIONS SECTION <<<<<
IBM XL C Enterprise Edition for AIX, Version 7.0.0.4 ---
***   Command Line Invocation ***
***   Options In Effect   ***

CPLUSCMT             LIST
64BIT
ALIAS=ANSI:NOTYPEPTR:NOALLPTRS:NOADDRTAKEN
LANGLVL=EXTC89:NOUCS

>>>>> FILE TABLE SECTION <<<<<
FILE CREATION        FROM
FILE NO   FILENAME                     DATE       TIME       FILE    LINE
      0   hello.c                      07/09/06   11:48:06
      1   /usr/include/stdio.h         11/04/05   20:51:47      0       1
      2   /usr/include/standards.h     03/17/05   04:13:32      1      43
      3   /usr/include/va_list.h       09/13/02   17:27:45      1     185
      4   /usr/include/sys/types.h     06/22/04   23:17:58      1     416
      5   /usr/include/strict_stdtypes.h
                                       03/15/03   16:50:03      4      54
      6   /usr/include/sys/inttypes.h 06/20/03   01:45:53      4      57
      7   /usr/include/stdint.h        07/29/04   15:37:58      6      51
      8   /usr/include/end_strict_stdtypes.h
                                       03/15/03   16:50:03      4      60
      9   /usr/include/sys/m_types.h   09/13/02   17:52:54      4     440
     10   /usr/include/sys/vm_types.h 05/24/04   16:29:58      9      40
     11   /usr/include/sys/limits.h    08/09/05   10:46:22      1     491
     12   /usr/include/float.h         08/24/04   05:53:30     11     285
```

`hello world` (assembler - 2)

```
…
>>>>> COMPILATION EPILOGUE SECTION <<<<<

IBM XL C/C++ Enterprise Edition for AIX Summary of Diagnosed Conditions

TOTAL   UNRECOVERABLE  SEVERE       ERROR      WARNING     INFORMATIONAL
(U)        (S)          (E)      (W)        (I)
    0            0          0           0          0              0


Source records read.......................................    3577

1501-008  Compilation successful for file hello.c. Object file created.


>>>>> OBJECT SECTION, NO OPTIMIZATION <<<<<


 GPR's set/used:   ss-s ssss ssss s---  ---- ---- ---- ---s
 FPR's set/used:   ssss ssss ssss ss--  ---- ---- ---- ----
 CCR's set/used:   ss-- -sss

…
```

`hello world` (assembler - 3)

```
     | 000000                          PDEF     main
    3|                                PROC
    0| 000000 mfspr    7C0802A6   1      LFLR     gr0=lr
    0| 000004 std      FBE1FFF8   1      ST8      #stack(gr1,-8)=gr31
    0| 000008 std      F8010010   1      ST8      #stack(gr1,16)=gr0
    0| 00000C stdu     F821FF81   1      ST8U     gr1,#stack(gr1,-128)=gr1
    0| 000010 ld       EBE20008   1      L8       gr31=.+CONSTANT_AREA(gr2,0)
    3|                          CL.1:
    6| 000014 addi     38600001   1      LI       gr3=1
    6| 000018 stw      90610070   1      ST4A     i(gr1,112)=gr3
    6| 00001C lwa      E8610072   1      L4A      gr3=i(gr1,112)
    6| 000020 cmpwi    2C03000A   2      C4       cr0=gr3,10
    6| 000024 bc       40800028   1      BF       CL.4,cr0,0x1/lt ,
    6|                          CL.3:
    8| 000028 addi     387F0008   1      AI       gr3=gr31,8
    8| 00002C bl       4BFFFFD5   0      CALL     gr3=printf,1,gr3,printf",gr1,cr[01567]",gr0",gr4"-gr12",fp0"-fp13"
    8| 000030 ori      60000000   1
    6| 000034 lwa      E8610072   1      L4A      gr3=i(gr1,112)
    6| 000038 addi     38630001   2      AI       gr3=gr3,1
    6| 00003C stw      90610070   1      ST4A     i(gr1,112)=gr3
    6| 000040 lwa      E8610072   1      L4A      gr3=i(gr1,112)
    6| 000044 cmpwi    2C03000A   2      C4       cr0=gr3,10
    6| 000048 bc       4180FFE0   1      BT       CL.3,cr0,0x1/lt ,
    6|                          CL.5:
    6|                          CL.4:
   10| 00004C addi     38600000   1      LI       gr3=0
   10| 000050 b        48000004   0      B        CL.6,-1
   10|                          CL.2:
   10|                          CL.6:
   10| 000054 ld       E8010090   1      L8       gr0=#stack(gr1,144)
   10| 000058 mtspr    7C0803A6   2      LLR      lr=gr0
   10| 00005C addi     38210080   1      AI       gr1=gr1,128
   10| 000060 ld       EBE1FFF8   1      L8       gr31=#stack(gr1,-8)
   10| 000064 bclr     4E800020   1      BA       lr
```

`hello world` (profiling)

- Profiling su IBM Power5, `printf()` non è la sola funzione invocata…

```
granularity: Each sample hit covers 4 bytes. No time has accumulated.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
  0.0       0.00     0.00        9     0.00     0.00  ._doprnt [1]
  0.0       0.00     0.00        9     0.00     0.00  ._xflsbuf [2]
  0.0       0.00     0.00        9     0.00     0.00  ._xwrite [3]
  0.0       0.00     0.00        9     0.00     0.00  .fwrite [4]
  0.0       0.00     0.00        9     0.00     0.00  .fwrite_unlocked [5]
  0.0       0.00     0.00        9     0.00     0.00  .memchr [6]
  0.0       0.00     0.00        9     0.00     0.00  .printf [7]
  0.0       0.00     0.00        9     0.00     0.00  .write [8]
  0.0       0.00     0.00        3     0.00     0.00  .splay [9]
  0.0       0.00     0.00        2     0.00     0.00  .free [10]
  0.0       0.00     0.00        2     0.00     0.00  .free_y [11]
  0.0       0.00     0.00        2     0.00     0.00  .free_y_heap [12]
  0.0       0.00     0.00        1     0.00     0.00  .__ioctl [13]
  0.0       0.00     0.00        1     0.00     0.00  ._findbuf [14]
  0.0       0.00     0.00        1     0.00     0.00  ._wrtchk [15]
  0.0       0.00     0.00        1     0.00     0.00  .exit [16]
  0.0       0.00     0.00        1     0.00     0.00  .ioctl [17]
  0.0       0.00     0.00        1     0.00     0.00  .isatty [18]
  0.0       0.00     0.00        1     0.00     0.00  .main [19]
  0.0       0.00     0.00        1     0.00     0.00  .moncontrol [20]
  0.0       0.00     0.00        1     0.00     0.00  .monitor [21]
  0.0       0.00     0.00        1     0.00     0.00  .pre_ioctl [22]
```

Interazioni

- Dirette:
  - CPU
  - memoria
  - alcuni dispositivi
- Indirette
  - gerarchia di memoria
  - contese per le risorse
  - *scheduling*
  - memoria virtuale
- Esplicitamente mediate dall'OS
  - I/O e comunicazioni in genere
  - richieste di servizi

I ferri del mestiere

- Strumenti di sviluppo
  - il compilatore tenta di produrre eseguibili ottimali
  - il risultato può essere inferiore alle aspettative
  - strumenti di analisi per capire cosa succede
  - librerie di soluzioni già ottimizzate
- Un'arte, non una scienza
  - esistono approcci, non soluzioni confezionate
  - ogni strumento ha pregi e difetti differenti
  - ogni codice ed ogni macchina ha le sue peculiarità
  - l'esperienza pratica è essenziale

Sommario

1. Cosa vuol dire *performance*
2. Concetti generali
3. **Algoritmi**
4. La gerarchia di memoria
5. Come usare al meglio la *cache*
6. Dal linguaggio alla CPU: il compilatore
7. Quanto costano le operazioni
8. Per riassumere
9. Il *profiling*
10. Librerie
11. L'ambiente circostante
12. Conclusioni

Sono algoritmi diversi? (I)

```
int c[N];
int i, j;

c[0] = 1;
c[1] = 0;
for(i = 2; i<N; ++i)
   for(c[i] = 0, j = i-1; !c[i] && j > 1; --j)
      c[i] |= i%j == 0;
```

```
int c[N];
int i, j;

c[0] = 1;
c[1] = 0;
for(i = 2; i<N; ++i)
   for(c[i] = 0, j = sqrt(i); !c[i] && j > 1; --j)
      c[i] |= i%j == 0;
```

Sono algoritmi diversi? (II)

```
int c[N];
int i, j;

c[0] = 1;
c[1] = 0;
for(i = 2; i<N; ++i)
   for(c[i] = 0, j = sqrt(i); !c[i] && j > 1; --j)
      c[i] |= i%j == 0;
```

```
int c[N];
int i, j;

c[0] = 1;
c[1] = 0;
for(i = 3; i<N; i+=2)
   for(c[i-1] = 1, c[i] = 0, j = sqrt(i); !c[i] && j > 1; --j)
      c[i] |= i%j == 0;
c[2] = 0;
```

Sono algoritmi diversi? (III)

```
int c[N];
int i, j;

c[0] = 1;
c[1] = 0;
for(i = 3; i<N; i+=2)
   for(c[i-1] = 1, c[i] = 0, j = sqrt(i); !c[i] && j > 1; --j)
      c[i] |= i%j == 0;
c[2] = 0;

int c[N];
int i, j;

bzero(c, sizeof(c));
c[0] = 1;
for(i = 2; i*i<N; ++i)
   if (!c[i])
     for(j = i; j*j < N; ++j)
      c[i*j] = 1;
```

Sono algoritmi diversi? (IIII)

```
int c[N];
int i, j;

bzero(c, sizeof(c));
c[0] = 1;
for(i = 2; i*i<N; ++i)
   if (!c[i])
     for(j = i; j*j < N; ++j)
      c[i*j] = 1;


char c[N];
int i, j;

bzero(c, sizeof(c));
c[0] = 1;
for(i = 2; i*i<N; ++i)
   if (!c[i])
     for(j = i; j*j < N; ++j)
      c[i*j] = 1;
```

Il commesso viaggiatore

Quale è il percorso minimo che passa per *n* città (una sola volta, arrivo = partenza)?

- **`n!`** permutazioni da controllare
  - 10! = 3628800
  - 15! = 1307674368000
- Complessità O(n!)

Prima ottimizzazione: eliminare le operazioni inutili → algoritmo

Strumenti:

- carta e penna
- strumenti di *profiling* (**`gprof`**)
- strumenti di *timing*

Esempio tratto da “*Algorithms alley*”, J Bentley, DDJ 04-05/99

Commesso viaggiatore: algoritmo 1

Primo algoritmo:

- calcolare il percorso di tutte le *n*! permutazioni
- *n*×*n*! distanze calcolate

Usando **`gprof`**:

```
7.6    151.28   14.05  39916800   0.00    0.00   .check1() [7]
6.0    162.35   11.07 439084800   0.00    0.00   .d(int,int) [8]
3.0    167.94    5.59 878169600   0.00    0.00   .sqr(float) [9]
1.7    171.08    3.14 439084800   0.00    0.00   .sqrt(double) [11]
```

- **`check1()`** (misura la distanza percorsa) chiamata *n*! volte
- **`d(,)`** (distanza tra due città) chiamata *n*!×*n* volte chiama:
  - **`sqrt()`** → **`1`** volta
  - **`sqr()`** → **`2`** volte
- È possibile ridurre il tempo totale di calcolo?

Commesso viaggiatore: algoritmo 2

Secondo algoritmo:
- percorso ABCD = percorso BCDA
- calcolare il percorso di (n – 1)! permutazioni
- nx(n -1)! distanze calcolate

Usando **`gprof`**:

```
7.2      13.76    1.20  3628800   0.00    0.00   .check1() [7]
5.4      14.67    0.91 39916800   0.00    0.00   .d(int,int) [8]
2.9      15.16    0.49 79833600   0.00    0.00   .sqr(float) [9]
1.5      15.41    0.25 39916800   0.00    0.00   .sqrt(double) [11]
```

- Le chiamate alle funzioni sono diminuite di un fattore **`n`**
- Il tempo per 11 città passa da 123'' a 11'' (11 volte meno)

Commesso viaggiatore: algoritmo 3

Terzo algoritmo:
- percorso ABCD → 3 distanze calcolate
- percorso EBCD → 3 distanze calcolate
- A(BCD)+E(BCD) → 4 distanze calcolate

Usando **`gprof`**:

```
4.3    5.32    0.28 13492900     0.00     0.00 .d(int,int) [7]
4.0    5.58    0.26        1   260.00 1210.00 .search3(int,double) [4]
3.0    5.78    0.20 12470600     0.00     0.00 .swap(int,int) [9]
2.1    5.92    0.14 26985800     0.00     0.00 .sqr(float) [10]
1.2    6.34    0.08 13492900     0.00     0.00 .sqrt(double) [14]
```

- La funzione **`d()`** é chiamata 3 volte di meno
  - le funzioni **`sqr()`** e **`sqrt()`** sono chiamate 3 volte di meno
- Il tempo per 11 città é passato a 4.6'' (26 volte più veloce)

Commesso viaggiatore: algoritmo 4

Quarto algoritmo:
- interrompo il calcolo se la distanza percorsa è maggiore del minimo già trovato

Usando **gprof**:

```
16.3       0.31    0.07                              .sqrt [6]
...
 4.7       0.39    0.02  1496674    0.00     0.00   .swap(int,int) [8]
 4.7       0.41    0.02   779743    0.00     0.00   .sqrt(double) [9]
 2.3       0.42    0.01   779743    0.00     0.00   .d(int,int) [7]
 0.0       0.43    0.00  1559486    0.00     0.00   .sqr(float) [13]
```

- La funzione `d()` è chiamata 17 volte in meno
  - le funzioni `sqr()` e `sqrt()` sono chiamate 17 volte di meno
- Il tempo per 11 città è passato a 0.29'' (425 volte più veloce)

- Può dipendere dalla configurazione di partenza!

Commesso viaggiatore: algoritmo 5

Quinto algoritmo:
- precalcolo le distanze: mi servono solo `n*n` distanze

Usando **gprof**:

```
31.2    0.11   0.05         1   50.00   90.00 .search5(int,double) [2]
25.0    0.15   0.04   1496674    0.00    0.00 .swap(int,int) [6]
 0.0    0.16   0.00       242    0.00    0.00 .sqr(float) [13]
 0.0    0.16   0.00       121    0.00    0.00 .sqrt(double) [24]
```

- Le funzioni `sqrt()` e `sqr()` sono calcolate solo *nxn* volte !!!
- Il tempo per 11 città è passato a 0.11'' (1100 volte più veloce)

Commesso viaggiatore: riepilogo

Primo algoritmo (forza bruta): tutti gli $n!$ percorsi
- $n \times n!$ distanze calcolate

Secondo algoritmo: solo $(n-1)!$ percorsi
- percorso ABCD = percorso BCDA
- $n \times (n-1)!$ distanze calcolate
- ABCD + EBCD = 6 distanze calcolate

Terzo algoritmo: calcolo incrementale dei percorsi
- da BCD → ABCD ed EBCD
- distanze calcolate << $n \times (n-1)!$
- A(BCD) + E(BCD) = 4 distanze calcolate

Quarto algoritmo: eliminazione dele false strade
- inutile continuare il calcolo se la distanza > distanza minima

Quinto algoritmo: riutilizzo dei calcoli
- precalcolo delle $n^2/2$ distanze tra i punti

Commesso viaggiatore: conclusioni

Tempi, in secondi, su IBM Power4 @ 1100 Mhz

Alg. \ Città	9	10	11	12	13	14	15
1	0.92	10.28	123.48	1583	-	-	-
2	0.10	1.03	11.22	134.07	-	-	-
3	-	0.46	4.59	50.50	605.99	-	-
4	-	-	0.29	1.50	11.29	98.73	-
5	-	-	0.11	0.57	4.29	37.56	287.83