CS3383 Lecture 1.2: The Master Theorem with applications

David Bremner

January 19, 2018

Contents

Generic divide and conquer algorithm

function $\textsc{Solve}$(P)
    if $|P|$ is small **then**
        SolveDirectly(P)
    else
        $P_1 \dots P_k = \text{Partition}(P)$
        for $i = 1 \dots k$ **do**
            $S_i = \text{Solve}(P_i)$
        end for
        Combine($S_1 \dots S_k$)
    end if
end function

- How many times do we recurse?

Generic divide and conquer algorithm

function $\textsc{Solve}$(P)
  if $|P|$ is small **then**
    SolveDirectly(P)
  else
    $P_1 \ldots P_k =$ Partition(P)
    for $i = 1 \ldots k$ **do**
      $S_i =$ Solve(P_i)
    end for
    Combine($S_1 \ldots S_k$)
  end if
end function

- How many times do we recurse?
- what fraction of input in each subproblem?

Generic divide and conquer algorithm

function $\text{SOLVE}(\text{P})$
  if $|P|$ is small **then**
    SolveDirectly(P)
  else
    $P_1 \ldots P_k = \text{Partition}(P)$
    for $i = 1 \ldots k$ **do**
      $S_i = \text{Solve}(P_i)$
    end for
    Combine($S_1 \ldots S_k$)
  end if
end function

- How many times do we recurse?
- what fraction of input in each subproblem?
- How much time to combine results?

Common recursive structure

A typical Divide and Conquer algorithm has

Common recursive structure

A typical Divide and Conquer algorithm has

- b : the branch factor, number of recursive calls per instantiation

Common recursive structure

A typical Divide and Conquer algorithm has

- b : the branch factor, number of recursive calls per instantiation
- s : the split, the inverse of the input size reduction (so recursing on $n/2$ would be $s = 2$)

Common recursive structure

A typical Divide and Conquer algorithm has

- b : the branch factor, number of recursive calls per instantiation
- s : the split, the inverse of the input size reduction (so recursing on $n/2$ would be $s = 2$)
- d : the degree of the polynomial of the running time of the find+combine steps

Common recursive structure

A typical Divide and Conquer algorithm has

- b : the branch factor, number of recursive calls per instantiation
- s : the split, the inverse of the input size reduction (so recursing on $n/2$ would be $s = 2$)
- d : the degree of the polynomial of the running time of the find+combine steps

Variations

Common recursive structure

A typical Divide and Conquer algorithm has

- b : the branch factor, number of recursive calls per instantiation
- s : the split, the inverse of the input size reduction (so recursing on $n/2$ would be $s = 2$)
- d : the degree of the polynomial of the running time of the find+combine steps

Variations

- e.g. one call of $\frac{1}{3}$ and one call of $\frac{2}{3}$,

Common recursive structure

A typical Divide and Conquer algorithm has

b : the branch factor, number of recursive calls per instantiation

s : the split, the inverse of the input size reduction (so recursing on $n/2$ would be $s = 2$)

d : the degree of the polynomial of the running time of the find+combine steps

Variations

- e.g. one call of $\frac{1}{3}$ and one call of $\frac{2}{3}$,
- partition+combine step $\Theta(n \log n)$.

The Master Theorem

If $\exists$ constants $b > 0$, $s > 1$ and $d \geq 0$ such that $T(n) = b \cdot T(\lceil \frac{n}{s} \rceil) + \Theta(n^d)$, then

The Master Theorem

If $\exists$ constants $b > 0$, $s > 1$ and $d \geq 0$ such that $T(n) = b \cdot T(\lceil \frac{n}{s} \rceil) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > log_s b \quad (\text{equiv. to } b < s^d) \\ \Theta(n^d \log n) & \text{if } d = log_s b \quad (\text{equiv. to } b = s^d) \\ \Theta(n^{\log_s b}) & \text{if } d < log_s b \quad (\text{equiv. to } b > s^d) \end{cases}$$

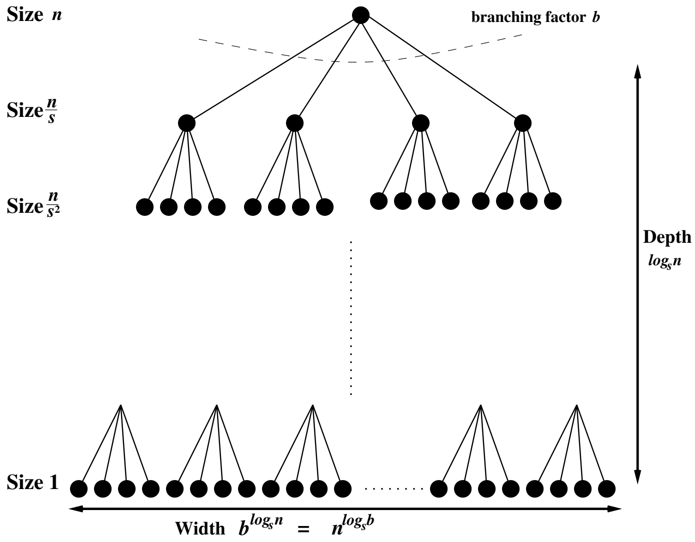
The Master Theorem

If $\exists$ constants $b > 0$, $s > 1$ and $d \geq 0$ such that $T(n) = b \cdot T(\lceil \frac{n}{s} \rceil) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > log_s b \quad (\text{equiv. to } b < s^d) \\ \Theta(n^d \log n) & \text{if } d = log_s b \quad (\text{equiv. to } b = s^d) \\ \Theta(n^{\log_s b}) & \text{if } d < log_s b \quad (\text{equiv. to } b > s^d) \end{cases}$$

A proof of this follows.

Proof of Master theorem, in pictures



Proof of Master Theorem

We assume w.l.o.g. n is an integer power of s. (If not, then what do we do?)

Proof of Master Theorem

We assume w.l.o.g. n is an integer power of s. (If not, then what do we do?)

The height of our recursion tree is $\log_s n$.

Proof of Master Theorem

We assume w.l.o.g. n is an integer power of s. (If not, then what do we do?)

The height of our recursion tree is $\log_s n$. At level i of the recursion tree (counting from 0) we have:

Proof of Master Theorem

We assume w.l.o.g. n is an integer power of s. (If not, then what do we do?)

The height of our recursion tree is $\log_s n$. At level i of the recursion tree (counting from 0) we have:

- the size of the data $= \frac{n}{s^i}$

Proof of Master Theorem

We assume w.l.o.g. n is an integer power of s. (If not, then what do we do?)

The height of our recursion tree is $\log_s n$. At level i of the recursion tree (counting from 0) we have:

- the size of the data $= \frac{n}{s^i}$
- the time for the combine step $= c \cdot \left(\frac{n}{s^i}\right)^d$

Proof of Master Theorem

We assume w.l.o.g. n is an integer power of s. (If not, then what do we do?)

The height of our recursion tree is $\log_s n$. At level i of the recursion tree (counting from 0) we have:

- the size of the data $= \frac{n}{s^i}$
- the time for the combine step $= c \cdot \left(\frac{n}{s^i}\right)^d$
- the number of recursive instantiations $= b^i$

Proof of Master Theorem

We assume w.l.o.g. n is an integer power of s. (If not, then what do we do?)

The height of our recursion tree is $\log_s n$. At level i of the recursion tree (counting from 0) we have:

- the size of the data $= \frac{n}{s^i}$
- the time for the combine step $= c \cdot \left(\frac{n}{s^i}\right)^d$
- the number of recursive instantiations $= b^i$

And so

$$T(n) = \sum_{i=0}^{\log_s n} c \cdot \left(\frac{n}{s^i}\right)^d \cdot b^i$$

Proof of Master theorem, $b = s^d$

$$T(n) = \sum_{i=0}^{\log_s n} c \cdot \left(\frac{n^d}{\left(s^d\right)^i} \right) \cdot b^i \; = \; c \cdot n^d \cdot \left(\sum_{i=0}^{\log_s n} \left(\frac{b}{s^d} \right)^i \right)$$

Proof of Master theorem, $b = s^d$

$$T(n) = \sum_{i=0}^{\log_s n} c \cdot \left(\frac{n^d}{\left(s^d\right)^i} \right) \cdot b^i \;=\; c \cdot n^d \cdot \left(\sum_{i=0}^{\log_s n} \left(\frac{b}{s^d} \right)^i \right)$$

If $b = s^d$, then

$$T(n) = c \cdot n^d \cdot \left(\sum_{i=0}^{\log_s n} 1 \right) \;=\; c \cdot n^d \log_s n$$

Proof of Master theorem, $b = s^d$

$$T(n) = \sum_{i=0}^{\log_s n} c \cdot \left(\frac{n^d}{\left(s^d\right)^i}\right) \cdot b^i \;=\; c \cdot n^d \cdot \left(\sum_{i=0}^{\log_s n} \left(\frac{b}{s^d}\right)^i\right)$$

If $b = s^d$, then

$$T(n) = c \cdot n^d \cdot \left(\sum_{i=0}^{\log_s n} 1\right) \;=\; c \cdot n^d \log_s n$$

so $T(n)$ is $\Theta(n^d \log n)$.

Proof of Master Theorem $b \neq s^d$ (1 of 2)

Otherwise ($b \neq s^d$), we have a geometric series,

$$T(n) = c \cdot n^d \cdot \left(\frac{\left(\frac{b}{s^d}\right)^{\log_s n+1} - 1}{\frac{b}{s^d} - 1} \right)$$

Proof of Master Theorem $b \neq s^d$ (1 of 2)

Otherwise ($b \neq s^d$), we have a geometric series,

$$T(n) = c \cdot n^d \cdot \left(\frac{\left(\frac{b}{s^d}\right)^{\log_s n+1} - 1}{\frac{b}{s^d} - 1} \right)$$

Applying $\dfrac{1}{p/q - 1} = \dfrac{q}{p - q}$

Proof of Master Theorem $b \neq s^d$ (1 of 2)

Otherwise ($b \neq s^d$), we have a geometric series,

$$T(n) = c \cdot n^d \cdot \left(\frac{\left(\frac{b}{s^d}\right)^{\log_s n+1} - 1}{\frac{b}{s^d} - 1} \right)$$

Applying $\dfrac{1}{p/q - 1} = \dfrac{q}{p - q}$

$$T(n) = \frac{s^d}{b - s^d} \cdot c \cdot n^d \cdot \left(\left(\frac{b}{s^d}\right)^{\log_s n+1} - 1 \right)$$

Proof of Master Theorem $b \neq s^d$ (1 of 2)

Otherwise ($b \neq s^d$), we have a geometric series,

$$T(n) = c \cdot n^d \cdot \left(\frac{\left(\frac{b}{s^d}\right)^{\log_s n+1} - 1}{\frac{b}{s^d} - 1} \right)$$

Applying $\dfrac{1}{p/q - 1} = \dfrac{q}{p-q}$

$$T(n) = \frac{s^d}{b - s^d} \cdot c \cdot n^d \cdot \left(\left(\frac{b}{s^d}\right)^{\log_s n+1} - 1 \right)$$

$$= \frac{s^d}{b - s^d} \cdot c \cdot n^d \cdot \left(\frac{b}{s^d}\right)^{\log_s n+1} - \frac{s^d}{b - s^d} \cdot c \cdot n^d$$

Proof of Master Theorem $b \neq s^d$ (2 of 2)

From rules of powers and logarithms:

$$\left(\frac{b}{s^d}\right)^{\log_s n+1} = \frac{b}{s^d} \cdot \left(\frac{b}{s^d}\right)^{\log_s n} = \frac{b}{s^d} \cdot \frac{b^{\log_s n}}{\left(s^d\right)^{\log_s n}}$$

Proof of Master Theorem $b \neq s^d$ (2 of 2)

From rules of powers and logarithms:

$$\left(\frac{b}{s^d}\right)^{\log_s n+1} = \frac{b}{s^d} \cdot \left(\frac{b}{s^d}\right)^{\log_s n} = \frac{b}{s^d} \cdot \frac{b^{\log_s n}}{\left(s^d\right)^{\log_s n}}$$

$$= \frac{b}{s^d} \cdot \frac{b^{\log_s n}}{n^d} = b \cdot \frac{n^{\log_s b}}{s^d n^d}$$

Proof of Master Theorem $b \neq s^d$ (2 of 2)

From rules of powers and logarithms:

$$\left(\frac{b}{s^d}\right)^{\log_s n+1} = \frac{b}{s^d} \cdot \left(\frac{b}{s^d}\right)^{\log_s n} = \frac{b}{s^d} \cdot \frac{b^{\log_s n}}{\left(s^d\right)^{\log_s n}}$$

$$= \frac{b}{s^d} \cdot \frac{b^{\log_s n}}{n^d} = b \cdot \frac{n^{\log_s b}}{s^d n^d}$$

Substituting in

$$T(n) = \frac{s^d n^d}{b - s^d} \cdot c \cdot \left(\frac{b}{s^d}\right)^{\log_s n+1} - \frac{s^d}{b - s^d} \cdot c \cdot n^d$$

Proof of Master Theorem $b \neq s^d$ (2 of 2)

From rules of powers and logarithms:

$$\left(\frac{b}{s^d}\right)^{\log_s n+1} = \frac{b}{s^d} \cdot \left(\frac{b}{s^d}\right)^{\log_s n} = \frac{b}{s^d} \cdot \frac{b^{\log_s n}}{\left(s^d\right)^{\log_s n}}$$

$$= \frac{b}{s^d} \cdot \frac{b^{\log_s n}}{n^d} = b \cdot \frac{n^{\log_s b}}{s^d n^d}$$

Substituting in

$$T(n) = \frac{s^d n^d}{b - s^d} \cdot c \cdot \left(\frac{b}{s^d}\right)^{\log_s n+1} - \frac{s^d}{b - s^d} \cdot c \cdot n^d$$

$$= \frac{b}{b - s^d} \cdot c \cdot n^{\log_s b} - \frac{s^d}{b - s^d} \cdot c \cdot n^d$$

Branching versus subproblem size

$$T(n) = \frac{b}{b - s^d} \cdot c \cdot n^{\log_s b} - \frac{s^d}{b - s^d} \cdot c \cdot n^d$$

Now we need to test b versus s^d.

Branching versus subproblem size

$$T(n) = \frac{b}{b - s^d} \cdot c \cdot n^{\log_s b} - \frac{s^d}{b - s^d} \cdot c \cdot n^d$$

Now we need to test b versus s^d.

If $b > s^d$ ($\log_s b > d$), first term dominates: $\Theta(n^{\log_s b})$.

Branching versus subproblem size

$$T(n) = \frac{b}{b - s^d} \cdot c \cdot n^{\log_s b} - \frac{s^d}{b - s^d} \cdot c \cdot n^d$$

Now we need to test b versus s^d.

If $b > s^d$ ($\log_s b > d$), first term dominates: $\Theta(n^{\log_s b})$.

If $b < s^d$ ($\log_s b < d$), then

$$T(n) = \frac{s^d}{s^d - b} \cdot c \cdot n^d - \frac{b}{s^d - b} \cdot c \cdot n^{\log_s b}$$

Branching versus subproblem size

$$T(n) = \frac{b}{b - s^d} \cdot c \cdot n^{\log_s b} - \frac{s^d}{b - s^d} \cdot c \cdot n^d$$

Now we need to test b versus s^d.

If $b > s^d$ ($\log_s b > d$), first term dominates: $\Theta(n^{\log_s b})$.

If $b < s^d$ ($\log_s b < d$), then

$$T(n) = \frac{s^d}{s^d - b} \cdot c \cdot n^d - \frac{b}{s^d - b} \cdot c \cdot n^{\log_s b}$$

new first term dominates: $\Theta(n^d)$.

Sanity check: Merge sort

Master Theorem

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > log_s b \\ \Theta(n^d \log n) & \text{if } d = log_s b \\ \Theta(n^{\log_s b}) & \text{if } d < log_s b \end{cases}$$

Sanity check: Merge sort

Master Theorem

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > log_s b \\ \Theta(n^d \log n) & \text{if } d = log_s b \\ \Theta(n^{\log_s b}) & \text{if } d < log_s b \end{cases}$$

Merge Sort

- $T(n) = bT(n/s) + \theta(n^d)$
- b how many recursive calls?
- s what is the the split (denominator of size)
- d degree

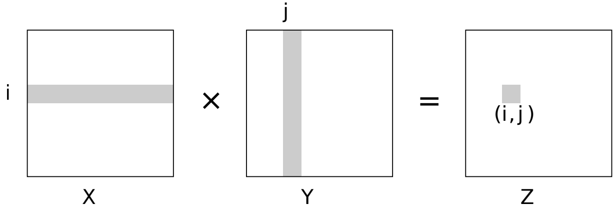
Contents

Matrix Multiplication

The product of two $n \times n$ matrices x and y is a third $n \times n$ matrix $Z = XY$, with

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$

where Z_{ij} is the entry in row i and column j of matrix Z.



Calculating Z directly using this formula takes $\Theta(n^3)$ time.

Matrix Multiplication: Blocks

decompose the input matrices into four blocks each (cutting the dimension n in half):

Matrix Multiplication: Blocks

decompose the input matrices into four blocks each (cutting the dimension n in half):

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Matrix Multiplication: Blocks

decompose the input matrices into four blocks each (cutting the dimension n in half):

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$\begin{aligned} XY &= \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} \\ &= \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix} \end{aligned}$$

Matrix Multiplication: Blocks

decompose the input matrices into four blocks each (cutting the dimension n in half):

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$= \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Eight subinstances $AE, BG, AF, BH, CE, DG, CF, DH$

Matrix Multiplication: Blocks

Recursing 8 times on subinstances of dimension $\frac{n}{2}$, and taking cn^2 time to add the results, gives the time recurrence:

$$T(n) \;=\; 8 \cdot T\left(\frac{n}{2}\right) + cn^2$$

Matrix Multiplication: Blocks

Recursing 8 times on subinstances of dimension $\frac{n}{2}$, and taking cn^2 time to add the results, gives the time recurrence:

$$T(n) \;=\; 8 \cdot T\left(\frac{n}{2}\right) + cn^2$$

Using the Master Theorem (and observing that $\log_2 8 = 3 > 2$) shows that this is a $\Theta(n^{\log_2 8}) \;=\; \Theta(n^3)$ algorithm.

Matrix Multiplication: Blocks

Recursing 8 times on subinstances of dimension $\frac{n}{2}$, and taking cn^2 time to add the results, gives the time recurrence:

$$T(n) \ = \ 8 \cdot T\left(\frac{n}{2}\right) + cn^2$$

Using the Master Theorem (and observing that $\log_2 8 = 3 > 2$) shows that this is a $\Theta(n^{\log_2 8}) \ = \ \Theta(n^3)$ algorithm.

So, just as with integer multiplication, the most direct way to split the instance does not produce an improvement in the running time.

Matrix Multiplication: Blocks

Recursing 8 times on subinstances of dimension $\frac{n}{2}$, and taking cn^2 time to add the results, gives the time recurrence:

$$T(n) \;=\; 8 \cdot T\left(\frac{n}{2}\right) + cn^2$$

Using the Master Theorem (and observing that $\log_2 8 = 3 > 2$) shows that this is a $\Theta(n^{\log_2 8}) \;=\; \Theta(n^3)$ algorithm.

So, just as with integer multiplication, the most direct way to split the instance does not produce an improvement in the running time.

(this is not technically "cubic algorithm", input size n^2.)

Matrix Multiplication: Strassen Decomposition

As with Integer Multiplication, we find we need a decomposition that reuses results.

Matrix Multiplication: Strassen Decomposition

As with Integer Multiplication, we find we need a decomposition that reuses results. Strassen found such a decomposition:

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

Matrix Multiplication: Strassen Decomposition

As with Integer Multiplication, we find we need a decomposition that reuses results. Strassen found such a decomposition:

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

where

$$\begin{aligned} P_1 &= A(F - H) & P_5 &= (A + D)(E + H) \\ P_2 &= (A + B)H & P_6 &= (B - D)(G + H) \\ P_3 &= (C + D)E & P_7 &= (A - C)(E + F) \\ P_4 &= D(G - E) & & \end{aligned}$$

Matrix Multiplication: Strassen Decomposition

This may not look like it would be an improvement since the decomposition is complicated, but in saving one recursive call, we get a time recurrence of

$$T(n) \;=\; 7 \cdot T\left(\frac{n}{2}\right) + cn^2$$

Matrix Multiplication: Strassen Decomposition

This may not look like it would be an improvement since the decomposition is complicated, but in saving one recursive call, we get a time recurrence of

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + cn^2$$

Using the Master Theorem (with $\log_2 7 > \log_2 4 = 2$) shows that this is a $\Theta(n^{\log_2 7})$ algorithm, approximately $\Theta(n^{2.81})$.

Matrix Multiplication: Strassen Decomposition

This may not look like it would be an improvement since the decomposition is complicated, but in saving one recursive call, we get a time recurrence of

$$T(n) \;=\; 7 \cdot T\left(\frac{n}{2}\right) + cn^2$$

Using the Master Theorem (with $\log_2 7 > \log_2 4 = 2$) shows that this is a $\Theta(n^{\log_2 7})$ algorithm, approximately $\Theta(n^{2.81})$.

Since the input size is $m = n^2$, the algorithm runs in approximately $\Theta(m^{1.404})$ time (versus the $\Theta(m^{1.5})$ of the original).