COMP202
Complexity of Algorithms

Divide & Conquer

Reading materials: Chapters 2.3.1 and 4 in CLRS

Learning Outcomes

At the conclusion of this set of lecture notes, you should:

1. Understand the general idea of Divide-and-Conquer algorithms.
2. Be familiar with some of the classical Divide-and-Conquer algorithms.
3. Be able to utilize the "Master Method" that may be used to derive closed form expressions for (some) recurrence relations.

Sorting

Sorting problem: Given a collection C of n elements (and a total ordering), arrange the elements of C into *non-decreasing* order, e.g.,

$$[45, 3, 67, 1, 5, 16, 105, 8] \longrightarrow \text{INPUT}$$
$$[1, 3, 5, 8, 16, 45, 67, 105] \longrightarrow \text{OUTPUT}$$

MergeSort, which we will now see, is a classical Divide-and-Conquer algorithm.

Incremental algorithms

We saw an algorithm for minimum finding last week, this was an *incremental* algorithm: having stored the minimum of $A[1..j-1]$, the next step finds and stores the minimum of $A[1..j]$.

The divide-and-conquer method is a means that can be used to solve some algorithmic problems. This general method consists of the following steps:

Divide-and-Conquer

The divide-and-conquer method is a means that can be used to solve some algorithmic problems. This general method consists of the following steps:

- *Divide*: If the input size is *small* then solve the problem directly; otherwise, divide the input data into two or more subsets, typically disjoint.

Divide-and-Conquer

The divide-and-conquer method is a means that can be used to solve some algorithmic problems. This general method consists of the following steps:

- *Divide*: If the input size is *small* then solve the problem directly; otherwise, divide the input data into two or more subsets, typically disjoint.
- *Recur*: *Recursively* solve the sub-problems associated with subsets.

Divide-and-Conquer

The divide-and-conquer method is a means that can be used to solve some algorithmic problems. This general method consists of the following steps:

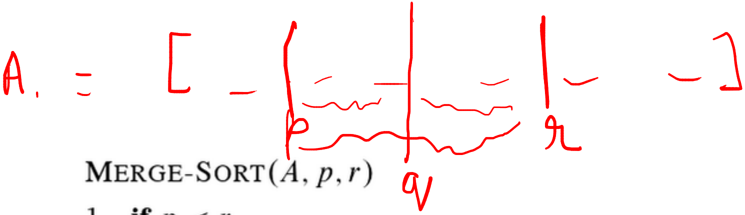
- *Divide*: If the input size is *small* then solve the problem directly; otherwise, divide the input data into two or more subsets, typically disjoint.
- *Recur*: *Recursively* solve the sub-problems associated with subsets.
- *Conquer*: Take the solutions to sub-problems and *merge* into a solution to the original problem.

MergeSort

MergeSort is one way we can apply the divide-and-conquer method to perform sorting. The MergeSort method consists of the following steps:

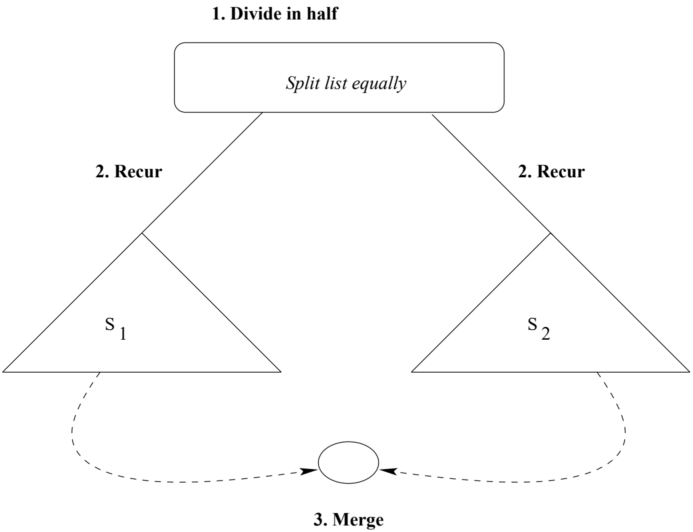
- *Divide*: If input sequence S has 0 or 1 elements, then return S; otherwise, split S into *two* sequences S_1 and S_2, each containing about $\frac{1}{2}$ of the elements in S.
- *Recur*: Recursively sort S_1 and S_2.
- *Conquer*: Put the elements back into S by merging the sorted sequences S_1 and S_2 into a *single sorted sequence*.

MergeSort: Pseudocode

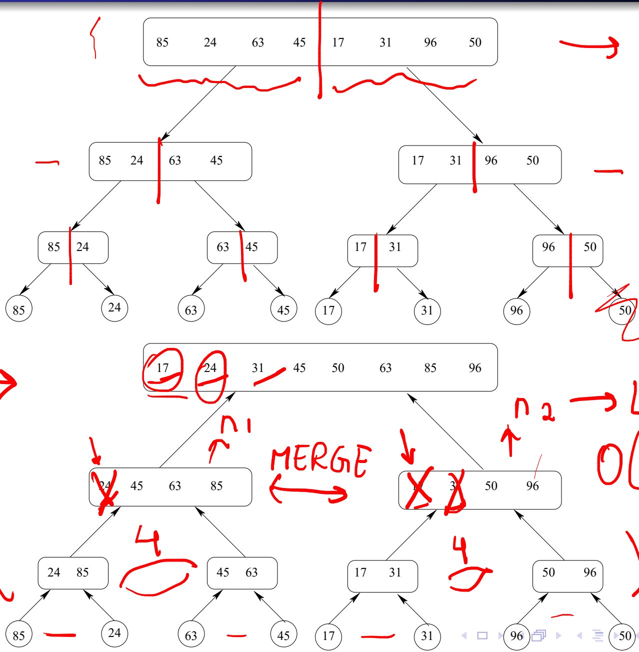



```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

MergeSort - Illustration



MergeSort - Example



1 → n

2

3

4

8 →

n_1

MERGE

n_2 → LENGTHS

$O(n_1+n_2)$

7 →

4

4

6 → → n

5

Theorem: *Merging* two sorted sequences S_1 and S_2 takes $O(n_1 + n_2)$ time, where n_1 is the size of S_1 and n_2 is the size of S_2.

MergeSort - Analysis

Theorem: *Merging* two sorted sequences S_1 and S_2 takes $O(n_1 + n_2)$ time, where n_1 is the size of S_1 and n_2 is the size of S_2.

Theorem: *MergeSort* runs in $O(n \log n)$ time in the *worst* (and *average*) case.

Theorem: *Merging* two sorted sequences S_1 and S_2 takes $O(n_1 + n_2)$ time, where n_1 is the size of S_1 and n_2 is the size of S_2.

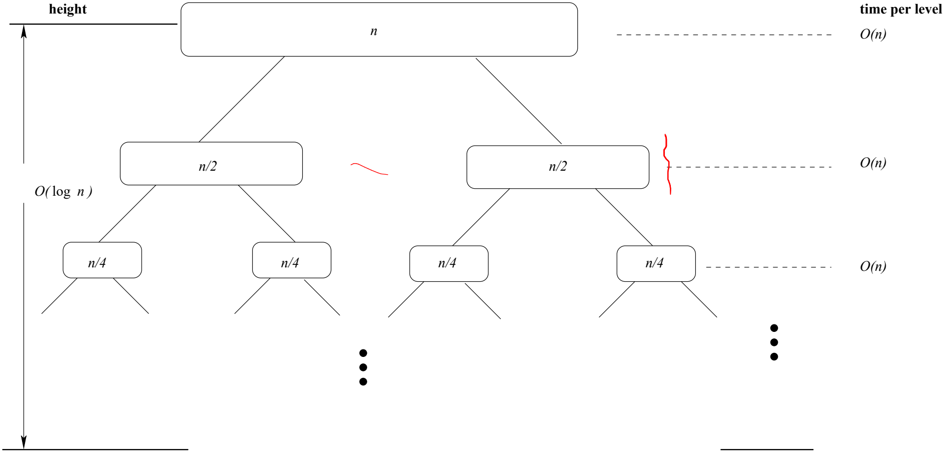
Theorem: *MergeSort* runs in $O(n \log n)$ time in the *worst* (and *average*) case.

Idea: During every recursive step, dividing each sublist takes time at most $O(n)$ time. Merging all of the lists in each level takes at most $O(n)$ time as well.

How many times do we recurse? This is at most $O(\log n)$ times. Hence this gives the $O(n \log n)$ running time for MergeSort.

$n, \frac{n}{2}, \frac{n}{2^2}, \ldots \frac{n}{2^k} = 1 \Rightarrow k = \log_2 n$

MergeSort - Analysis



Total time: $O(n \log n)$

Divide-and-Conquer (cont.)

To *analyze* the running time of a Divide-and-Conquer algorithm we typically utilize a *recurrence relation*, where

- $T(n)$ denotes the worst-possible running time on an input of size n.

We then want to characterize $T(n)$ using an equation that relates $T(n)$ to values of function T for problem sizes smaller than n.

MergeSort recurrence

For example, the running time of the MergeSort algorithm can be written in this form:

$$T(n) = \begin{cases} c' \longrightarrow 1 & \text{if } n < 2 \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

where c is a small constant that represents how much work is done for the comparisons for merging the lists, etc.

For example, the running time of the MergeSort algorithm can be written in this form:

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

where c is a small constant that represents how much work is done for the comparisons for merging the lists, etc.

(**Note**: Strictly speaking this should be written as

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn & \text{otherwise} \end{cases}$$

but in terms of the *asymptotic running time*, the "floors" and "ceilings" are unnecessary, and this can be formally proven.)

Binary search recurrence

$A = [\quad \downarrow \geq 10? \quad]$ SORTED

For another example, the running time of the binary search method (on sorted arrays) can be described as

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ \underline{T(n/2)} + \underline{c} & \text{otherwise} \end{cases}$$

where the constant c here represents the work needed to determine the midpoint of the sublist and do the comparison to determine if the sought for value has been found (or updating the bottom/top index to determine the next sublist to search).

$T(n) = O(\log n)$

Substitution Method

One way to solve a *Divide-and-Conquer* recurrence equation is to use the *iterative substitution method*, a.k.a. "plug-and-chug", e.g.

$2^{\log_2 n} = n$

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ 2T(n/2) + cn & \text{otherwise.} \end{cases}$$

$$T\left(\frac{n}{2}\right) = \begin{cases} c & 2 \le \frac{n}{2} \\ 2T\left(\frac{n}{4}\right) + c \cdot \frac{n}{2} \end{cases}$$

Assuming that n is a power of 2 (for simplicity) we get

$$T(n) = 2\left[2T(n/2^2) + c(n/2)\right] + cn = 2^2 T(n/2^2) + 2cn$$

$$= 4T\left(\frac{n}{4}\right) + 2cn$$

After j iterations we have

$$T(n) = 2^j T(n/2^j) + jcn.$$

$= 1 \equiv j = \log_2 n.$

For $j = \log_2 n$, we get

$$T(n) = 2^{\log_2 n} \cdot c + cn \log_2 n = cn + cn \log_2 n = \Theta(n \log n).$$

Maximum-subarray problem

Say you want to invest in a company's stock. You are allowed to buy one unit of the stock on a day of your choosing, and sell it at a later day of your choosing.

Maximum-subarray problem

Say you want to invest in a company's stock. You are allowed to buy one unit of the stock on a day of your choosing, and sell it at a later day of your choosing.

Say you also have the power to see future stock prices. Goal is to maximise profit (sell price - buy price).

Say you want to invest in a company's stock. You are allowed to buy one unit of the stock on a day of your choosing, and sell it at a later day of your choosing.

Say you also have the power to see future stock prices. Goal is to maximise profit (sell price - buy price).

$106 - 63 = 43$

$$\text{Price} = [100, 113, 110, 85, 105, 102, 86, 63, 81, 101, 94, 106, 101]$$
$$\text{Change} = [13, -3, -25, 20, -3, 16, -23, 18, 20, -7, 12, -5]$$

$18 + 20 - 7 + 12$
$= 43$

Maximum-subarray problem

Say you want to invest in a company's stock. You are allowed to buy one unit of the stock on a day of your choosing, and sell it at a later day of your choosing.

Say you also have the power to see future stock prices. Goal is to maximise profit (sell price - buy price).

$$\text{Price} = [100, 113, 110, 85, 105, 102, 86, 63, 81, 101, 94, 106, 101]$$
$$\text{Change} = [13, -3, -25, 20, -3, 16, -23, 18, 20, -7, 12, -5]$$

Brute force: Try every possible buy and sell dates in which buy precedes sell date. A period of n days has $\binom{n}{2}$ such dates. Since $\binom{n}{2} = \Theta(n^2)$, and the evaluation of each pair of dates can take $O(1)$ time at best, this approach takes $\Omega(n^2)$ time.

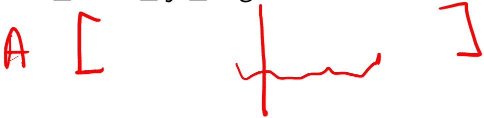
Divide-and-conquer solution

Suppose we want to find a maximum subarray of the subarray $A[low..high]$. Let mid be the midpoint of the array. Any contiguous subarray $A[i..j]$ of $A[low..high]$ must lie in exactly one of the following places:

Divide-and-conquer solution

Suppose we want to find a maximum subarray of the subarray $A[low..high]$. Let mid be the midpoint of the array. Any contiguous subarray $A[i..j]$ of $A[low..high]$ must lie in exactly one of the following places:

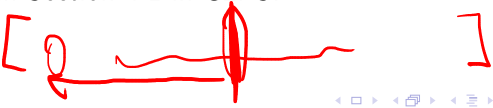
- entirely in $A[low..mid]$, i.e., $low \leq i \leq j \leq mid$,
- entirely in $A[mid+1..high]$, i.e., $mid < i \leq j \leq high$, or
- crossing the midpoint, i.e., $low \leq mid \leq j \leq high$.



Suppose we want to find a maximum subarray of the subarray $A[low..high]$. Let mid be the midpoint of the array. Any contiguous subarray $A[i..j]$ of $A[low..high]$ must lie in exactly one of the following places:

- entirely in $A[low..mid]$, i.e., $low \leq i \leq j \leq mid$,
- entirely in $A[mid+1..high]$, i.e., $mid < i \leq j \leq high$, or
- crossing the midpoint, i.e., $low \leq mid \leq j \leq high$.

Will find maximum subarray values of each of these three: the first two recursively because they are smaller instances of the original problem. For the third, can find maximum subarrays of the form $A[i..mid]$ and $A[mid+1..j]$ in linear time and combine them. See pseudo-code in Section 4.1 in CLRS!



Analysis

As in MergeSort, let $T(n)$ denote running time of this algorithm on an array of size n.

Clearly, $T(1) = \Theta(1)$. The first two bullets on the previous slide contribute to $2T(n/2)$, and the third to cn for a constant c. Thus,

$$T(n) = \begin{cases} \cancel{\Theta(n)} \rightarrow O(1) & n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$$

This is the same recurrence as MergeSort! So the running time of this algorithm is $O(n \log n)$.

EX. SHEET: $O(n)$

ALG.

Matrix Multiplication: Another example

Suppose we are given two $n \times n$ matrices X and Y, and we wish to compute their product $Z = X \cdot Y$, which is defined in this manner:

$X = (\ )_{n \times n}$, $Y = (\ )_{n \times n}$

$$Z[i,j] = \sum_{k=1}^{n} X[i,k] \cdot Y[k,j].$$

Each element $Z[i,j]$ takes time $\Theta(n)$ to compute (since there are n multiplications and $n-1$ additions to perform), assuming that any multiplication or addition takes constant time.

As there are n^2 elements to compute for Z, this naturally leads to an $\Theta(n^3)$ time algorithm.

See pseudocode in Section 4.2, CLRS!

Matrix Multiplication (cont.)

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae+bg & - \\ -- & -- \end{pmatrix}$$

For example, we have the following products

$$\begin{pmatrix} 2 & 5 \\ -1 & 3 \end{pmatrix} \begin{pmatrix} 1 & -2 \\ 3 & 0 \end{pmatrix} = \begin{pmatrix} 17 & -4 \\ 8 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 3 & 4 \\ 0 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 2 \\ -1 & 2 & 0 \\ 4 & 5 & -2 \end{pmatrix} = \begin{pmatrix} 15 & 28 & -4 \\ -5 & -3 & 2 \\ 4 & 8 & 0 \end{pmatrix}.$$

A, B, C, D, X

E, F, G, H, Y

$2 - 3 + 16 = 15$

$2 + 6 + 20 = 28$

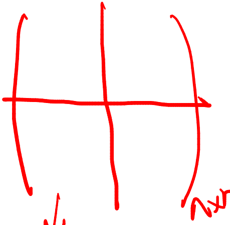
$n^2 \cdot n = O(n^3)$

entries

ALG.

Another useful way of viewing this product (especially for large values of n) is in terms of products of *sub-matrices*

$$\begin{pmatrix} I & J \\ K & L \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$



where

$$4\left(\frac{n}{2}\right)^2$$

$$\begin{aligned} I &= AE + BG \\ J &= AF + BH \\ K &= CE + DG \\ L &= CF + DH. \end{aligned}$$

EACH IS $\left(\frac{n}{2} \times \frac{n}{2}\right)$

This gives a *Divide-and-Conquer* algorithm with running time $T(n)$, such that $T(n) = 8T(n/2) + bn^2 = \Theta(n^3)$ (using the Master Method, which we'll see soon).

Strassen's Algorithm for matrix multiplication

Volker Strassen (in 1969) realized that matrix multiplication can be performed using a divide and conquer approach that uses a *smaller* number of multiplications.

First define the following seven matrix products:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}\begin{pmatrix} E & F \\ G & H \end{pmatrix}_{n \times n} = \begin{pmatrix} I & J \\ K & L \end{pmatrix}$$

1 MATRIX MULT. EACH

$$\begin{aligned}
S_1 &= A(F-H)\\
S_2 &= (A+B)H\\
S_3 &= (C+D)E\\
S_4 &= D(G-E)\\
S_5 &= (A+D)(E+H)\\
S_6 &= (B-D)(G+H)\\
S_7 &= (C-A)(E+F)
\end{aligned}$$

$J = AF + BH$

$AF - \cancel{AH}$

$\cancel{AH} + BH$

Strassen's Algorithm (cont.)

We can then find I, J, K, and L using the S_i's as follows:

$$
\begin{aligned}
I &= S_4 + S_5 + S_6 - S_2 \\
J &= S_1 + S_2 \\
K &= S_3 + S_4 \\
L &= S_1 + S_5 + S_7 - S_3
\end{aligned}
$$

Strassen's Algorithm (cont.)

Thus, we can compute $Z = X \cdot Y$ using only *seven* recursive multiplications of matrices of size $n/2 \times n/2$.
Therefore, the running time of Strassen's Algorithm satisfies this recursive relation:

$$T(n) = 7\,T(n/2) + bn^2.$$

Strassen's Algorithm (cont.)

Thus, we can compute $Z = X \cdot Y$ using only *seven* recursive multiplications of matrices of size $n/2 \times n/2$.
Therefore, the running time of Strassen's Algorithm satisfies this recursive relation:

$T(1) = 1$

$$T(n) = 7\,T(n/2) + bn^2.$$

Using this recursive formula, together with the Master Method (which we'll see soon), we obtain:

Theorem: We can multiply two $n \times n$ matrices in $O(n^{\log_2 7}) = O(n^{2.808})$ time.

$n^{2.30}$ --- BEST KNOWN.

Divide-and-Conquer style recurrence relations

$$T(n) = 2\underbrace{T\left(\frac{n}{2}\right)} + cn$$

We saw an example of using the Substitution method ("plug-and-chug") to solve the recurrence for MergeSort.

Divide-and-Conquer style recurrence relations

We saw an example of using the Substitution method ("plug-and-chug") to solve the recurrence for MergeSort.

We might also "guess" a solution to a divide-and-conquer style recurrence relation, and then formally prove this using the method of *mathematical induction*. WARNING

We saw an example of using the Substitution method ("plug-and-chug") to solve the recurrence for MergeSort.

We might also "guess" a solution to a divide-and-conquer style recurrence relation, ~~and~~ then formally prove this using the method of *mathematical induction*.

Let us consider the more general case, where

$$T(n) = \begin{cases} c & \text{if } n \leq d \\ aT(n/b) + f(n) & \text{if } n > d \end{cases}$$

for some constants a, b, c, and d, and some function $f(n)$.

MERGE SORT $a = 2, \; b = 2,$
$f(n) = cn.$

We can "unwind" this recurrence relation in this manner:

$$T(n) = aT(n/b) + f(n)$$

$$T\left(\frac{n}{b}\right) = aT\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)$$

Divide-and-Conquer style recurrence relations (cont.)

We can "unwind" this recurrence relation in this manner:

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ &= a\left[aT(n/b^2) + f(n/b)\right] + f(n) \end{aligned}$$

Divide-and-Conquer style recurrence relations (cont.)

We can "unwind" this recurrence relation in this manner:

$$
\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a\left[aT(n/b^2) + f(n/b)\right] + f(n) \\
&= a^2 T(n/b^2) + af(n/b) + f(n)
\end{aligned}
$$

Divide-and-Conquer style recurrence relations (cont.)

We can "unwind" this recurrence relation in this manner:

$$
\begin{aligned}
T(n) &= \boxed{aT(n/b) + f(n)} \\
&= a\left[aT(n/b^2) + f(n/b)\right] + f(n) \\
&= \boxed{a^2T(n/b^2) + af(n/b) + f(n)} \\
&= a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \\
&= a^4T(n/b^4) + a^3f(n/b^3) + a^2f(n)/b^2) + af(n/b) + f(n) \\
&= \ldots
\end{aligned}
$$

$$\frac{n}{b^j} = 1 \equiv b^j = n$$

$$j \log b = \log n$$

Divide-and-Conquer style recurrence relations (cont.)

We can "unwind" this recurrence relation in this manner:

$$
\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a\left[aT(n/b^2) + f(n/b)\right] + f(n) \\
&= a^2T(n/b^2) + af(n/b) + f(n) \\
&= a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \\
&= a^4T(n/b^4) + a^3f(n/b^3) + a^2f(n)/b^2) + af(n/b) + f(n) \\
&= \ldots \\
&= \underbrace{a^{\log_b n}}_{= n^{\log_b a}} T(1) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)
\end{aligned}
$$

$j = \log_b n$

$b^j = n$

Divide-and-Conquer style recurrence relations (cont.)

We can "unwind" this recurrence relation in this manner:

$$\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a\left[aT(n/b^2) + f(n/b)\right] + f(n) \\
&= a^2T(n/b^2) + af(n/b) + f(n) \\
&= a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \\
&= a^4T(n/b^4) + a^3f(n/b^3) + a^2f(n)/b^2) + af(n/b) + f(n) \\
&= \dots \\
&= a^{\log_b n}T(1) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \\
&= n^{\log_b a}T(1) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j).
\end{aligned}$$

(We use $a^{\log_b n} = n^{\log_b a}$ to get the final line.)

Recurrence tree for $T(n) = 3T(n/4) + cn^2$

I/P SIZE RECURRENCE TREE

$cn^2 \cdot \left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \cdots\right)$

$O(1)$

n

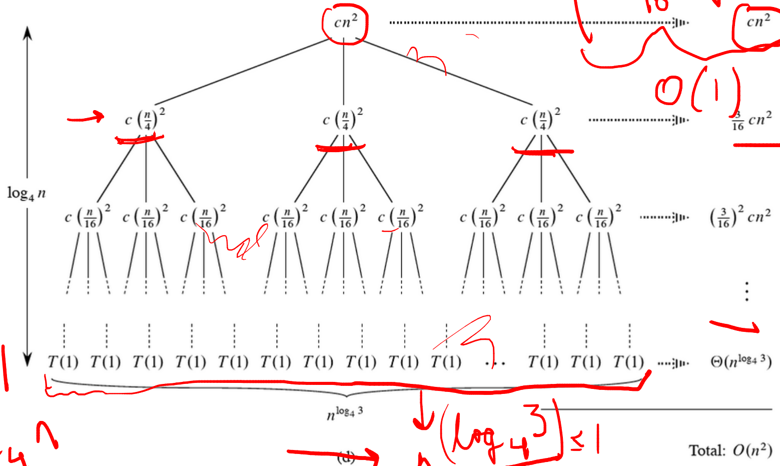
$n/4$

$n/4^2$

$\vdots$

$n/4^i = 1$

$\log_4 n$



$n^{(\log_4 3)} \le 1$

Divide-and-Conquer style recurrence relations (cont.)

With a careful analysis of the previous expression, we can make some conclusions about the (asymptotic) form of $T(n)$.

With a careful analysis of the previous expression, we can make some conclusions about the (asymptotic) form of $T(n)$.

AUX UNIT-COST STEPS

For example, if $f(n)$ is "smaller" than $\boxed{n^{\log_b a}}$ (in a suitable fashion), then the first term is going to dominate the summation, and we will have $T(n) = \Theta(n^{\log_b a})$.

With a careful analysis of the previous expression, we can make some conclusions about the (asymptotic) form of $T(n)$.

For example, if $f(n)$ is "smaller" than $n^{\log_b a}$ (in a suitable fashion), then the first term is going to dominate the summation, and we will have $T(n) = \Theta(n^{\log_b a})$.

If, on the other hand, we have $n^{\log_b a}$ is "smaller" than $f(n)$ (and the function f is "nice"), then the summation will dominate this expression and we will have $T(n) = \Theta(f(n))$.

With a careful analysis of the previous expression, we can make some conclusions about the (asymptotic) form of $T(n)$.

For example, if $f(n)$ is "smaller" than $n^{\log_b a}$ (in a suitable fashion), then the first term is going to dominate the summation, and we will have $T(n) = \Theta(n^{\log_b a})$.

If, on the other hand, we have $n^{\log_b a}$ is "smaller" than $f(n)$ (and the function f is "nice"), then the summation will dominate this expression and we will have $T(n) = \Theta(f(n))$.

And there's a third case to consider too...

The Master Method

We make the following claim:

$\boxed{T(n) = aT\left(\frac{n}{b}\right) + f(n)}$

$T(n) = 2T\left(\frac{n}{2}\right) + cn \rightarrow a = 2, b = 2 \quad \log_b a = 1$

$f(n) = \Theta(n)$

Theorem

Suppose that $T(n)$ and $f(n)$ satisfy the recurrence relation defined previously (for some constants $a, b, \delta,$ and d).
Then

1. *If there is a constant $\varepsilon > 0$ such that $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$.*
2. *If there is a constant $k \geq 0$ such that $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$.*
3. *If there are constants $\varepsilon > 0$ and $\delta < 1$ such that $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$ and $a\, f(n/b) \leq \delta\, f(n)$ for $n > d$, then $T(n)$ is $\Theta(f(n))$.* ✓

Examples using the Master Method

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Consider these examples:

1. $T(n) = 4T(n/2) + n$.

Examples using the Master Method

$T(1) = 1$ $\quad n^{\log_b a}$ vs. $f(n)$

Consider these examples:

1. $T(n) = 4T(n/2) + n.$

   $a = 4,\ b = 2,\ f(n) = n$

   In this case we have $n^{\log_b a} = n^{\log_2 4} = n^2$. Therefore we are in Case 1, since $f(n) = n = O(n^{2-\varepsilon})$ for $\varepsilon = 1$. Thus, we conclude that $T(n)$ is $\Theta(n^2)$ by the Master Method.

   $n^2 = O(n^2 \log n)$

Examples using the Master Method

Consider these examples:

1. $T(n) = 4T(n/2) + n$.

   In this case we have $n^{\log_b a} = n^{\log_2 4} = n^2$. Therefore we are in Case 1, since $f(n) = n = O(n^{2-\varepsilon})$ for $\varepsilon = 1$. Thus, we conclude that $T(n)$ is $\Theta(n^2)$ by the Master Method.

2. $T(n) = 2T(n/2) + cn$. (This is the MergeSort type of recurrence relation.)

   $a=2$, $b=2$, $f(n)=cn$

   Here we have $n^{\log_b a} = n^{\log_2 2} = n^1 = n$. Hence we are in Case 2 with $k = 0$ (why?), so that $T(n)$ is $\Theta(n \log n)$.

Consider these examples:

1. $T(n) = 4T(n/2) + n$.

   In this case we have $n^{\log_b a} = n^{\log_2 4} = n^2$. Therefore we are in Case 1, since $f(n) = n = O(n^{2-\varepsilon})$ for $\varepsilon = 1$. Thus, we conclude that $T(n)$ is $\Theta(n^2)$ by the Master Method.

2. $T(n) = 2T(n/2) + \underline{cn}$. (This is the MergeSort type of recurrence relation.)

   Here we have $n^{\log_b a} = n^{\log_2 2} = n^1 = n$. Hence we are in Case 2 with $k = 0$ (why?), so that $T(n)$ is $\Theta(n \log n)$.

3. $T(n) = 2T(n/2) + \underline{n \log n}$.

   $a = b = 2, \quad f(n) = \underline{n(\log n)}$

   As before, we have $\log_2 2 = 1$, so $n^{\log_b a} = \underline{n}$. We are also in Case 2, with $k = 1$ as $f(n)$ is $\Theta(n \log n)$.
   Therefore, we conclude $T(n)$ is $\Theta(n \log^2 n)$.

$a \quad b \quad f(n)$

n^2 vs $n^{2.5}$

4. $T(n) = 9T(n/3) + n^{2.5}$.

   In this case we have $n^{\log_b a} = n^{\log_3 9} = n^2$.

   $a f\left(\frac{n}{b}\right) \le \delta \cdot f(n)$

   We are in Case 3, since $f(n) = n^{2.5} = \Omega(n^{2+\varepsilon})$, for $\varepsilon = 1/2$, $\forall n \ge d$

   and $a f(n/b) = 9(n/3)^{2.5} = (1/3)^{1/2} f(n)$. (So $\delta = (1/3)^{1/2}$ works for the statement of Case 3.) This means that $T(n)$ is $\Theta(n^{2.5})$ by the Master Method.

$$9 \cdot \frac{n^{2.5}}{3^{2.5}} = \cancel{3}^{\cancel{2}} \cdot \frac{n^{2.5}}{3^{\cancel{2}.5}} = \frac{n^{2.5}}{3^{.5}}$$

Examples using the Master Method (cont.)

$\log_3 3 = 1, \log_3 7, \log_3 9 = 2$

4. $T(n) = 9T(n/3) + n^{2.5}$.

   In this case we have $n^{\log_b a} = n^{\log_3 9} = n^2$.
   We are in Case 3, since $f(n) = n^{2.5} = \Omega(n^{2+\varepsilon})$, for $\varepsilon = 1/2$, and $a\,f(n/b) = 9(n/3)^{2.5} = (1/3)^{1/2} f(n)$. (So $\delta = (1/3)^{1/2}$ works for the statement of Case 3.) This means that $T(n)$ is $\Theta(n^{2.5})$ by the Master Method.

5. $T(n) = 7T(n/3) + n$. $a = 7$, $b = 3$, $f(n) = n$

   Here we see that $n^{\log_b a} = n^{\log_3 7}$. Since $1 < \log_3 7$, we see that there is some (small) $\varepsilon > 0$ such that $1 < \log_3 7 - \varepsilon$ and therefore $f(n) = n = O(n^{\log_3 7 - \varepsilon})$. Hence, we see that $T(n)$ is $\Theta(n^{\log_3 7})$.

$a \qquad \cancel{b}\ \sqrt{n}$

$\shortparallel$

6. **Variable change**: $T(n) = 2T(n^{1/2}) + \log n$.
Unfortunately this form doesn't allow us to use the master method.
Fix: Set $k = \log n$. $\longleftrightarrow$ $2^k = n$ $\qquad \sqrt{n} = \sqrt{2^k} = 2^{k/2}$

$$T(n) = T(2^k) = 2T(2^{k/2}) + k.$$

Substituting into $S(k) = T(2^k)$, we get

$$S(k) = 2S(k/2) + k.$$

$a = 2,\ b = 2,\ f(k) = k$

Master method gives $S(k) = O(k \log k)$. Substituting back for $T(n)$ implies $T(n) = O(\log n \log \log n)$.

Now we consider a Divide-and-Conquer method to multiply integers (especially when the integers are very large, such as those used in various encryption schemes).

Suppose that I and J are two integers with n binary digits each.

$$\{0,1\} \qquad \{0,1\}$$

$$(I_1 \quad I_2 \;---\; I_n)$$

$$J_1 \quad J_2 \;---\; J_n$$

$$I \times J_n$$

$$I \times J_{n-1} \quad 0$$

Fast multiplication of integers

Now we consider a Divide-and-Conquer method to multiply integers (especially when the integers are very large, such as those used in various encryption schemes).

Suppose that I and J are two integers with n binary digits each.

We can easily find $I + J$ and $I - J$ in time $\Theta(n)$. Computing the product $I \cdot J$, using the common grade-school algorithm, take $\Theta(n^2)$ time. Can we do better? (Assuming that n is quite large...)

Fast multiplication of integers

Now we consider a Divide-and-Conquer method to multiply integers (especially when the integers are very large, such as those used in various encryption schemes).

Suppose that I and J are two integers with n binary digits each.

We can easily find $I + J$ and $I - J$ in time $\Theta(n)$. Computing the product $I \cdot J$, using the common grade-school algorithm, take $\Theta(n^2)$ time. Can we do better? (Assuming that n is quite large...)

We will assume now that n is a power of 2 (which we can do by padding I and J with zeroes at the beginning).

Taking an inspiration (maybe?) from the MergeSort algorithm, let us consider writing I and J in two parts, namely the high-order and low-order bits.

$I = \underbrace{3182}\,\underbrace{4763}$

$J = \underbrace{1234}\,\underbrace{5678}$

In other words, we can write I and J as

$$
\begin{aligned}
I &= I_h 2^{n/2} + I_\ell \\
J &= J_h 2^{n/2} + J_\ell
\end{aligned}
$$

$I_h = 3182$

$I_h \cdot 10^4 \quad 3182\,0000$

$I_\ell \quad + \quad 4763$

I

Taking an inspiration (maybe?) from the MergeSort algorithm, let us consider writing I and J in two parts, namely the high-order and low-order bits.

In other words, we can write I and J as

$$\begin{aligned} I &= I_h 2^{n/2} + I_\ell \\ J &= J_h 2^{n/2} + J_\ell \end{aligned}$$

We observe that multiplying an integer by a power of 2 (like 2^k) is easy to do on a computer, as it is a (left) shift operation.

If a shift by one bit takes constant time to perform, then multiplying by 2^k takes $\Theta(k)$ time.

Fast multiplication of integers (cont.)

Given the representation of I and J above, in terms of the high- and low-order bits, let's concentrate on computing the product of I and J.

Fast multiplication of integers (cont.)

Given the representation of I and J above, in terms of the high- and low-order bits, let's concentrate on computing the product of I and J.

Given the above representation, we see that

$$\begin{aligned} I \cdot J &= (I_h 2^{n/2} + I_\ell) \cdot (J_h 2^{n/2} + J_\ell) \\ &= I_h J_h 2^n + I_\ell J_h 2^{n/2} + I_h J_\ell 2^{n/2} + I_\ell J_\ell \end{aligned}$$

Fast multiplication of integers (cont.)

Given the representation of I and J above, in terms of the high- and low-order bits, let's concentrate on computing the product of I and J.

Given the above representation, we see that

$$\begin{aligned} I \cdot J &= (I_h 2^{n/2} + I_\ell) \cdot (J_h 2^{n/2} + J_\ell) \\ &= I_h J_h 2^n + I_\ell J_h 2^{n/2} + I_h J_\ell 2^{n/2} + I_\ell J_\ell \end{aligned}$$

We can use a Divide-and-Conquer type of algorithm to compute this product.

Fast multiplication of integers (cont.)

Given the representation of I and J above, in terms of the high- and low-order bits, let's concentrate on computing the product of I and J.

Given the above representation, we see that

$$\begin{aligned} I \cdot J &= (I_h 2^{n/2} + I_\ell) \cdot (J_h 2^{n/2} + J_\ell) \\ &= I_h J_h 2^n + I_\ell J_h 2^{n/2} + I_h J_\ell 2^{n/2} + I_\ell J_\ell \end{aligned}$$

We can use a Divide-and-Conquer type of algorithm to compute this product.

Namely, let's divide I and J into the high- and low-order bits as above, compute four products of these (size $n/2$ bit) pieces, and then combine them together as above (performing some shift operations, to multiply by $2^{n/2}$ or 2^n where needed).

Fast multiplication of integers (cont.)

Computing the shifts and additions requires, in total, time $O(n)$ so we end up with a recurrence relation that looks like (for $n \geq 2$)

$$T(n) = 4T(n/2) + cn$$

a b $f(n)$

$n^{\log_b a}$

$\|$

$n^{\log_2 4} = n^2$

n^2 vs cn

for some constant c.

Fast multiplication of integers (cont.)

Computing the shifts and additions requires, in total, time $O(n)$ so we end up with a recurrence relation that looks like (for $n \geq 2$)

$$T(n) = 4T(n/2) + cn$$

for some constant c.

Applying the Master Method, we find we are in Case 1, namely since $n^{\log_2 4} = n^2$, and $f(n) = cn = O(n^{2-\varepsilon})$ for, say, $\varepsilon = 1/2$, we find that $T(n) = \Theta(n^2)$.

Fast multiplication of integers (cont.)

Computing the shifts and additions requires, in total, time $O(n)$ so we end up with a recurrence relation that looks like (for $n \geq 2$)

$$T(n) = 4T(n/2) + cn$$

for some constant c.

Applying the Master Method, we find we are in Case 1, namely since $n^{\log_2 4} = n^2$, and $f(n) = cn = O(n^{2-\varepsilon})$ for, say, $\varepsilon = 1/2$, we find that $T(n) = \Theta(n^2)$.

Hmmm, so our Divide-and-Conquer method hasn't helped us do things any quicker...

Fast multiplication of integers (cont.)

The Master Method, however, gives us insight into how we could improve our approach.

Fast multiplication of integers (cont.)

The Master Method, however, gives us insight into how we could improve our approach.

If we can reduce the *number* of recursive calls (by reducing the value of $a = 4$), we could improve the overall running time.

The Master Method, however, gives us insight into how we could improve our approach.

If we can reduce the *number* of recursive calls (by reducing the value of $a = 4$), we could improve the overall running time.

$I = I_h \cdot 2^{n/2} + I_\ell$

So let us consider the product

$$(I_h - I_\ell)(J_\ell - J_h) = \underline{I_h J_\ell} - I_h J_h - I_\ell J_\ell + \underline{I_\ell J_h}.$$

The Master Method, however, gives us insight into how we could improve our approach.

If we can reduce the *number* of recursive calls (by reducing the value of $a = 4$), we could improve the overall running time.

So let us consider the product

$$(I_h - I_\ell)(J_\ell - J_h) = I_h J_\ell - I_h J_h - I_\ell J_\ell + I_\ell J_h.$$

This might be an odd product to consider, but note that it consists of two of the products we want to compute, namely

$$I_h J_\ell \text{ and } I_\ell J_h,$$

together with two of the products that we (will) compute recursively, namely

$$I_h J_h \text{ and } I_\ell J_\ell.$$

Fast multiplication of integers (cont.)

So our Divide-and-Conquer strategy becomes this:

1. Write I and J in terms of their high- and low-order bits.

Fast multiplication of integers (cont.)

So our Divide-and-Conquer strategy becomes this:

1. Write I and J in terms of their high- and low-order bits.
2. Compute (recursively) the products $I_h J_h$, $I_\ell J_\ell$, and $(I_h - I_\ell)(J_\ell - J_h)$.

Fast multiplication of integers (cont.)

So our Divide-and-Conquer strategy becomes this:

1. Write I and J in terms of their high- and low-order bits.
2. Compute (recursively) the products $I_h J_h$, $I_\ell J_\ell$, and $(I_h - I_\ell)(J_\ell - J_h)$.
3. Combine these together to get

$$\begin{aligned} I \cdot J &= (I_h 2^{n/2} + I_\ell)(J_h 2^{n/2} + J_\ell) \\ &= I_h J_h 2^n + [(I_h - I_\ell)(J_\ell - J_h) + I_h J_h + I_\ell J_\ell] 2^{n/2} + I_\ell J_\ell \\ &= I_h J_h 2^n + I_\ell J_h 2^{n/2} + I_h J_\ell 2^{n/2} + I_\ell J_\ell . \end{aligned}$$

Fast multiplication of integers (cont.)

So our Divide-and-Conquer strategy becomes this:

1. Write I and J in terms of their high- and low-order bits.
2. Compute (recursively) the products $I_h J_h$, $I_\ell J_\ell$, and $(I_h - I_\ell)(J_\ell - J_h)$.
3. Combine these together to get

$$\begin{aligned} I \cdot J &= (I_h 2^{n/2} + I_\ell)(J_h 2^{n/2} + J_\ell) \\ &= I_h J_h 2^n + [(I_h - I_\ell)(J_\ell - J_h) + I_h J_h + I_\ell J_\ell] 2^{n/2} + I_\ell J_\ell \\ &= I_h J_h 2^n + I_\ell J_h 2^{n/2} + I_h J_\ell 2^{n/2} + I_\ell J_\ell . \end{aligned}$$

The important point is that this requires only <u>three</u> recursive multiplications of integers with $n/2$ bits.

Fast multiplication of integers (cont.)

The recurrence relation describing the running time to multiply the integers now becomes

$$T(n) = 3T(n/2) + cn$$

for a constant $c > 0$.

The recurrence relation describing the running time to multiply the integers now becomes

$$T(n) = \underset{a}{3}\,T(n/\underset{b}{2}) + \underset{f(n)}{cn}$$

$n^{\log_2 3}$

for a constant $c > 0$.

Since $1 < \log_2 3$, we can find a small $\varepsilon > 0$ such that $f(n) = cn = O(n^{\log_2 3 - \varepsilon})$. So we're still in Case 1 of the Master Method, but now we have this result:

The recurrence relation describing the running time to multiply the integers now becomes

FFT: $O\left(\frac{n \log n}{\log \log n}\right)$

$$T(n) = 3T(n/2) + cn$$

for a constant $c > 0$.

$\Omega(n)$

Since $1 < \log_2 3$, we can find a small $\varepsilon > 0$ such that $f(n) = cn = O(n^{\log_2 3 - \varepsilon})$. So we're still in Case 1 of the Master Method, but now we have this result:

Theorem

We can multiply two n-bit integers in time $\Theta(n^{\log_2 3}) = O(n^{1.585})$.

HARVEY, VAN DER HOEVEN '19: $O(n \log n)$