INT202
Complexity of Algorithms
Fundamental Techniques

XJTLU/SAT/INT

SEM2 AY2020-2021

Office Hours

Lecturer:

Rui Yang

Thursday 11pm-1pm

Office: SD529

email: R.Yang@xjtlu.edu.cn

Teaching Assistants:

Jingwei Guo

Wednesday 2pm-4pm

Office: EE511

email: Jingwei.Guo19@student.xjtlu.edu.cn

Peisong Li

Monday 2pm-4pm

Office: EE513

email: Peisong.Li20@student.xjtlu.edu.cn

Divide-and-Conquer

We have already discussed the divide-and-conquer method when we talked about sorting. To remind you, here is the general outline for using this method:

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

Divide-and-Conquer

To analyze the running time of a divide-and-conquer algorithm we typically utilize a *recurrence relation*, where $T(n)$ denotes the 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, e.g.

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

Substitution Method

In the iterative substitution, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$
\begin{aligned}
T(n) &= 2\boxed{T(n/2)} + bn \\
&= 2(\boxed{2T(n/2^2)) + b(n/2)}) + bn \\
&= 2^2 T(n/2^2) + 2bn \\
&= 2^3 T(n/2^3) + 3bn \\
&= 2^4 T(n/2^4) + 4bn \\
&= \ldots \\
&= 2^i T(n/2^i) + ibn
\end{aligned}
$$

Note that base, T(n)=b, case occurs when $2^i=n$. That is, i = log n.

So, $T(n) = bn + bn\log n$

Thus, T(n) is O(n log n).

The Master Method

It is a "cook-book" method to solve

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

wherein $d \geq 1, a > 0, c > 0, b > 1$

The Master Method

It is a "cook-book" method to solve

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

The Master Method (cont.)

In the **The Master Theorem**:

1. Case 1: applies where $f(n)$ is polynomially smaller than the special function $n^{\log_b a}$.

2. Case 2: applies where $f(n)$ is asymptotically close to the special function $n^{\log_b a}$.

3. Case 3: applies where $f(n)$ is polynomially larger than the special function $n^{\log_b a}$.

*f(n) is polynomially smaller than g(n) if $f(n)=O(g(n)/n^{\epsilon})$ for some $\epsilon>0$.

*f(n) is polynomially larger than g(n) if $f(n)=\Omega(g(n)n^{\epsilon})$ for some $\epsilon>0$

1. If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ∎

The Master Method (cont.)

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

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Example 5.7: *Consider the recurrence*

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

In this case, $n^{\log_b a} = n^{\log_2 4} = n^2$. *Thus, we are in Case 1, for* $f(n)$ *is* $O(n^{2-\varepsilon})$ *for* $\varepsilon = 1$. *This means that* $T(n)$ *is* $\Theta(n^2)$ *by the master method.*

The Master Method (cont.)

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

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

$$T(n) = T(2n/3) + 1,$$

in which $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(\lg n)$.

The Master Method (cont.)

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

1. If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Example 5.9: *Consider the recurrence*

$$T(n) = T(n/3) + n,$$

which is the recurrence for a geometrically decreasing summation that starts with n. In this case, $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$. Thus, we are in Case 3, for $f(n)$ is $\Omega(n^{0+\varepsilon})$, for $\varepsilon = 1$, and $af(n/b) = n/3 = (1/3)f(n)$. This means that $T(n)$ is $\Theta(n)$ by the master method.

The Master Method (cont.)

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

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

$$T(n) = 2T(n/2) + n \lg n ,$$

even though it appears to have the proper form: $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n$. You might mistakenly think that case 3 should apply, since $f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a} = n$. The problem is that it is not *polynomially* larger. The ratio $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ is asymptotically less than n^{ϵ} for any positive constant ϵ.

The Master Method (cont.)

Example 5.11: *Finally, consider the recurrence*

$$T(n) = 2T(n^{1/2}) + \log n.$$

Unfortunately, this equation is not in a form that allows us to use the master method. We can put it into such a form, however, by introducing the variable $k = \log n$, *which lets us write*

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

Substituting into this the equation $S(k) = T(2^k)$, *we get that*

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

Now, this recurrence equation allows us to use master method, which specifies that $S(k)$ *is* $O(k \log k)$. *Substituting back for* $T(n)$ *implies* $T(n)$ *is* $O(\log n \log \log n)$.

Matrix Multiplication: An example

Question Suppose we are given two $n \times n$ matrices X and Y, and we wish to compute their product Z = XY, which is defined so that

$$Z[i,j] = \sum_{k=0}^{n-1} X[i,k] \cdot Y[k,j]$$

which is an equation that immediately gives rise to a simple $O(n^3)$ time algorithm.

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

Matrix Multiplication: An example

Submatrices Suppose n is a power of two and let us partition X, Y, and Z each into four $(n/2)\times(n/2)$ matrices, so that we can rewrite Z = XY as

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

Thus,

$$I = AE + BG$$
$$J = AF + BH$$
$$K = CE + DG$$
$$L = CF + DH$$

Matrix Multiplication: An example

Divide-and-conquer algorithm computes $Z = XY$ by computing I, J, K, and L from the subarrays A through G.

$$I = AE + BG$$
$$J = AF + BH$$
$$K = CE + DG$$
$$L = CF + DH$$

By the above equations, we can compute I, J, K, and L from the eight recursively computed matrix products on $(\mathrm{n}/2)\times(\mathrm{n}/2)$ subarrays, plus four additions that can be done in $O(n^2)$ time.

Thus, the above set of equations give rise to a divide-and-conquer algorithm whose running time T(n) is characterized by the recurrence

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

for some constant b > 0.

This equation implies: T(n) = $O(n^3)$ by the master theorem.

Matrix Multiplication: An example

Strassen's algorithm organises arithmetic involving the subarrays A through G so that we can compute I,J, K, and L using just seven recursive matrix multiplications (by Volker Strassen 1969).

Define seven submatrix products:

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

Given these seven submatrix products, we can compute I,J, K, and L as

$$
\begin{aligned}
I &= S_5 + S_6 + S_4 - S_2 = AE + BG. \\
J &= S_1 + S_2 = AF + BH. \\
K &= S_3 + S_4 = CE + DG. \\
L &= S_1 - S_7 - S_3 + S_5 = CF + DH.
\end{aligned}
$$

Matrix Multiplication: An example

Strassen's algorithm organises arithmetic involving the subarrays A through G so that we can compute I,J, K, and L using just seven recursive matrix multiplications.

Given these seven submatrix products, we can compute I,J, K, and L as

$$I = S_5 + S_6 + S_4 - S_2 = AE + BG.$$
$$J = S_1 + S_2 = AF + BH.$$
$$K = S_3 + S_4 = CE + DG.$$
$$L = S_1 - S_7 - S_3 + S_5 = CF + DH.$$

Thus, we can compute $Z = XY$ using seven recursive multiplications of matrices of size $(n/2)\times(n/2)$. Thus, we characterize the running time $T(n)$ as

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

for some constant $b > 0$.

By the master theorem, we can multiply two n x n matrices in $O(n^{\log 7})$ time using Strassen's algorithm.

Matrix Multiplication: An example

$$Z[i,j] = \sum_{k=0}^{n-1} X[i,k] \cdot Y[k,j]$$

The exponent of matrix multiplication: smallest number ω such that for all ε>0 $O(n^{\omega+\varepsilon})$ operations suffice

- Standard algorithm ω ≤ 3
- Strassen (1969) ω < 2.81
- Pan (1978) ω < 2.79
- Bini et al. (1979) ω < 2.78
- Schönhage (1981) ω < 2.55
- Pan; Romani; Coppersmith + Winograd (1981-1982) ω < 2.50
- Strassen (1987) ω < 2.48
- Coppersmith + Winograd (1987) ω < 2.375
- Stothers (2010) ω < 2.3737
- Williams (2011) ω < 2.3729
- Le Gall (2014) ω < 2.37286

Counting inversions: Another example

This example is inspired by (if not directly related to) some of the "ranking systems" that are becoming more popular on some websites.

Suppose that you've rated a set of films or books (for example). In particular, you've rated n films by ranking them from your most favorite (ranked at 1) to least favorite (ranked at n).

In order to give a recommendation to you, this website wants to compare your ratings of these films with those of other people (for the same films) to see how similar they are.

How can you do this?

Counting inversions (cont.)

In other words, how can you compare your rankings

1 2 3 4 5 6 7 8 9 10

to another ranking

2 7 10 4 6 1 3 9 8 5?

Or even to another person's rankings

8 9 10 1 3 4 2 5 6 7?

Which one of these is "closest" to your rankings?

Counting inversions (cont.)

One proposed way of measuring the similarity is to count the number of *inversions*.

Suppose that

$$a_1, a_2, a_3, \ldots, a_n$$

denotes a permutation of the integers $1, 2, \ldots, n$. The pair of integers i, j are said to form an inversion if $i < j$, but $a_i > a_j$.

(We can generalize this idea to any sequence of distinct integers.)

We will count the number of inversions to measure the similarity of someone's rankings to yours.

Counting inversions (cont.)

For example, the permutation

$$1\ 2\ 4\ 3$$

contains one inversion (the 4 and the 3), while the permutation

$$1\ 4\ 3\ 2$$

has three (the 3*,* 4 pair, the 2*,* 3 and the 2*,* 4 pair).

In other words, to find the number of inversions, we count the pairs $i \neq j$ that are *out of order* in the permutation.

Counting inversions (cont.)

The number of inversions can range from 0, for the permutation

$$1\ 2\ 3\ .\ .\ .\ n,$$

up to $\binom{n}{2} = \frac{n(n-1)}{2}$ for the permutation $C_n^m = \frac{A_n^m}{m!} = \frac{n!}{m!\,(n-m)!}$

$$n\ \ n-1\ .\ .\ .\ 2\ \ 1.$$

Other examples:

2 1 3 4 5 has one inversion,

2 3 4 5 1 has four inversions.

Counting inversions: How do we do it?

So how do we count the number of inversions in a given permutation of n numbers?

The "naive" approach is to check all $\binom{n}{2}$ pairs to see if they form an inversion in the permutation. This gives an algorithm with $O(n^2)$ running time.

Can we do better?

Claim: We can count inversions using a divide-and-conquer algorithm that runs in time O(n log n).

A divide-and-conquer way to count inversions

Idea:

As with similar divide-and-conquer algorithms, we divide the permutation into two (nearly equal) parts. Then we (recursively) count the number of inversions in each part.

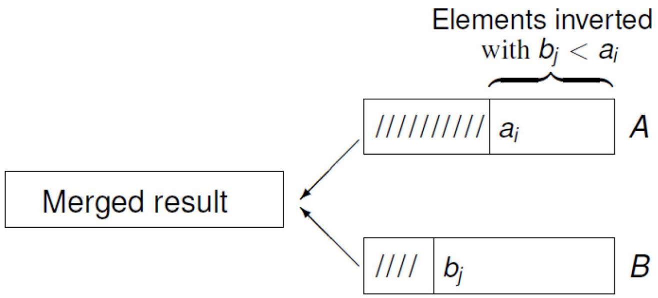
This gives us most of the inversions. We then need to get the number of inversions that involve one element of the first list, and one element of the second.

To do that we *sort* each sublist and merge them into a single (sorted) list. As we merge them together into a single list, we can count the inversions from such pairs mentioned above.
In other words, we're performing a modified MergeSort!

Divide-and-conquer for counting inversions

Suppose that we've divided the list into *A* (the first half) and *B* (the second half) and have counted the inversions in each.

After sorting them, the idea for counting the additional inversions is as follows:



As we merge the lists, every time we take an element from the list *B*, it forms an inversion with all of the *remaining* (unused) elements in list *A*.

A recursive algorithm for counting inversions

COUNT**I**NVERSIONS(L)

▷ Input: A list, L, of distinct integers.
▷ Output: The number of inversions in L.
if L has one element in it **then**
  there are no inversions, so Return $(0, L)$
else
  ▷ Divide the list into two halves
    A contains the first $\lfloor n/2 \rfloor$ elements
    B contains the last $\lceil n/2 \rceil$ elements
  $(k_A, A) = \text{CountInversions}(A)$
  $(k_B, B) = \text{CountInversions}(B)$
  $(k, L) = \text{MergeAndCount}(A, B)$
  Return $(k_A + k_B + k, L)$

The MERGEANDCOUNT method

MERGE**A**ND**C**OUNT(A, B)

$Current_A \leftarrow 0$
$Current_B \leftarrow 0$
$Count \leftarrow 0$
$L \leftarrow$ empty list
while both lists (A and B) are non-empty
    Let a_i and b_j denote the elements pointed to by $Current_A$ and $Current_B$.
    Append the smaller of a_i and b_j to L.
    if b_j is the smaller element **then**
        Increase $Count$ by the number of elements remaining in A.
    Advance the $Current$ pointer of the appropriate list.
Once one of A and B is empty, append the remaining elements to L.
Return ($Count$, L)

Counting inversions - The payoff

As mentioned earlier, this method for counting inversions is basically a modified version of the MergeSort algorithm.

Hence, we can count the number of inversions in a permutation in time $O(n \log n)$.

In terms of the ranking system describe earlier, the number of inversions for a permutation is a measure of how "out of order"

it is as compared to the identity permutation

$$1\ 2\ 3\ .\ .\ .\ n$$

and hence could be used to measure the "similarity" to the identity permutation.