Divide & Conquer

**Observation 1.** Often a problem \( P \) of size \( n \) can be solved by dividing it into several small problems of the same type and appropriately combining their solutions to solve the original problem.

These algorithms all have similar form

**Pre-work (Divide):** Transform problem \( P \) of size \( n \) into problems \( P_1, \ldots, P_a \) of sizes \( n_1, \ldots, n_a \) (each \( n_i < n \))

**Work:** Solve each of \( P_1, \ldots, P_a \)

**Post-work (Conquer):** Appropriately combine solutions of \( P_1, \ldots, P_a \) to construct solution of \( P \)

This structure suggests estimating the worst-case run time \( T(n) \) for a problem of size \( n \) by

\[
T(n) \leq \text{time to solve sub-problems} + \text{time to do pre- and post-work}
\]

\[
T(n) \leq \sum_{i=1}^{a} T(n_i) + f(n)
\]

Frequently all of the \( n_i \) are the same, say size \( n/b \), yielding

\[
T(n) \leq aT(n/b) + f(n)
\]

Last time we saw several examples

- **Huffman Encoding** \( T(n) \leq T(n-1) + c \log n \)
- **Selection Sort** \( T(n) \leq T(n-1) + cn \)
- **Merge Sort** \( T(n) \leq 2T(n/2) + cn \)
- **Binary Search** \( T(n) \leq T(n/2) + c \)
- **n-Digit Number Multiplication** \( T(n) \leq 4T(n/2) + cn \)

Solving Recurrences

All of the recurrences presented have one of the two forms

- \( T(n) \leq T(n-a) + f(n) \), where \( a \in \mathbb{N} \), or
- \( T(n) \leq aT(n^b) + f(n) \), where \( a, b \in \mathbb{N} \). [In fact, \( b = 2 \) in all of our examples so far.]

Thinking about Recurrence Relations: The Work Tree

Consider the Merge Sort recurrence \( T(n) \leq cn + 2T(n/2) \).

If we drew a tree of all of the calls to MergeSort, we might be able to count operations.

- **Level 0:** Initial call: at most \( cn \) steps to merge two sorted lists of length \( n/2 \) plus
- **Level 1:** at most \( c(n/2) + c(n/2) \) steps to merge 2 pairs of sorted lists of size \( n/4 \)
- **Level 2:** at most \( 4cn/4 \) steps to merge 4 pairs of sorted lists of size \( n/8 \)
Level $(\log n) - 1$: at most $n/2c2$ steps to merge $n/2$ pairs of sorted lists of size 2.

The total amount of work at each level is $cn$ and there are $\log n$ levels, giving $O(n \log n)$ steps in total.

We can rephrase the total amount of work as

$$T(n) \leq \sum_{i=0}^{k} \left( \text{number of recursive calls at level } i \right) \times \left( \text{number of steps performed} \right)$$

If we are splitting into $a$ problems of size $n/b$ at each recursive call and performing $f(n)$ steps of pre- and post-work with every call, we get

$$T(n) \leq \sum_{i=0}^{\log_b n} a^i \times f(n/b^i)$$

The behavior of this sum clearly depends upon the relation among $a$, $b$, and $f()$.

**The Master Theorem**

**Theorem 1.** If $T(n) \leq aT([n/b]) + O(n^d)$ for some positive constants $a$, $b$, $d$ then

$$T(n) = \begin{cases} 
O(n^d) & (a < b^d) \\
O(n^d \log n) & (a = b^d) \\
O(n^{\log_b a}) & (a > b^d)
\end{cases}$$

In fact:

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

**Proof.** We will assume that $n$ is a power of $b$. This does not influence the final bound and it allows us to ignore the ceiling. Imagine that we draw a tree expressing where the work is getting done.

Note: $a^{\log_b n} = n^{\log_b a}$.

- The size of the subproblems decrease by a factor of $b$ with each level of the recursion, so the tree has height $\log_b(n)$.
- A node in the tree does $O((n/b^k)^d)$ work at level $k$.
- There are $a^k$ nodes at level $k$, so the total work done at level $k$ is $a^k \times O\left(\left(\frac{n}{b^k}\right)^d\right) = O(n^d) \times \left(\frac{a}{b^d}\right)^k$
- As $k$ goes from 0 to $\log_b(n)$, the amount of work forms a geometric series with the ratio $\frac{a}{b^d}$.
- $T(n) \leq \sum_{i=0}^{\log_b n} a^i \times c(n/b^i)^d = cn^d \sum_{i=0}^{\log_b n} \frac{a^i}{(b^i)^d} = cn^d \sum_{i=0}^{\log_b n} (\frac{a}{b^d})^i = cn^d((\frac{a}{b^d})^{1+\log_b n} - 1)/(\frac{a}{b^d} - 1)$
- If this ratio is smaller than 1 then the first term $O(n^d)$ dominates. If the ratio is larger than 1, then the final term $n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\right) = O(n^{\log_b a})$ dominates. If the ratio is one, then there are $O(\log n)$ terms, each with $O(n^d)$ work. These cases correspond to the theorem.
So, we have a useful theorem! If we had tried it on \( T(n) \leq cn + 4T\left(\frac{n}{2}\right) \), we would have seen that \( a = 4, \ b = 2, \ d = 1 \) gives \( a > b^d \), so \( T(n) = O(n^{\log_2 4}) = O(n^2) \) and we could have quit at that point.

Suppose though, in our integer multiplication work, we could achieve \( T(n) \leq cn + 3T\left(\frac{n}{2}\right) \), that is, only 3 multiplications and a linear number of additions. Then we’d have \( a = 3, \ b = 2, \ d = 1 \) giving \( a > b^d \) and \( T(n) = O(n^{\log_2 3}) = O(n^{1.58}) \).

So, let’s see if we can do that!

Recall that we have two numbers \( A_1 = x_1 2^{n/2} + y_1 \) and \( A_2 = x_2 2^{n/2} + y_2 \).

So \( A_1 A_2 = (x_1 2^{n/2} + y_1)(x_2 2^{n/2} + y_2) = x_1 x_2 2^n + (x_1 y_2 + x_2 y_1) 2^{n/2} + y_1 y_2 \).

Consider \( x_1 y_2 + x_2 y_1 \).

Suppose I could replace it by an equivalent value of the form \( x_1 x_2 + y_1 y_2 - PQ \), where \( PQ \) is a single multiplication of \( n/2 \)-bit integers. Then we are only performing 3 such multiplication!

What would \( PQ \) be? Well \( PQ = x_1 y_2 + x_2 y_1 - x_1 x_2 - y_1 y_2 \). But \( x_1 y_2 + x_2 y_1 - x_1 x_2 - y_1 y_2 = (x_1 - y_1)(y_2 - x_2) \).

So, let \( P = x_1 - y_1 \) and \( Q = y_2 - x_2 \), then \( A_1 A_2 = x_1 x_2 2^n + PQ 2^{n/2} + y_1 y_2 \), requiring only three multiplications of \( n/2 \)-bit integers and \( O(n) \) steps of additional work to construct \( A_1 A_2 \) from these three components.

But this algorithm isn’t the end of the story on multiplying large integers....

**Theorem 2** (Karatsuba 1962). *Any two \( n \)-bit integers can be multiplied in time \( O(n^{1.58}) \).*

**Theorem 3** (Schnhage and Strassen 1971). *Any two \( n \)-bit integers can be multiplied in time \( O(n \log n \log \log n) \).*

More recently

**Theorem 4** (Furer 2007). *Any two \( n \)-bit integers can be multiplied in time \( O(n \log n) 2^{O(\log^* n)} \).*

And, in late-breaking news

**Theorem 5** (Harvey and van der Hoeven 2018). *Any two \( n \)-bit integers can be multiplied in time \( O(n \log n) 2^{2\log^* n} \).*

**Matrix Multiplication**

**INPUT:** Two \( n \times n \) matrices \( X \) and \( Y \) with real-valued entries.

**OUTPUT:** \( Z = X \times Y \)

Recall the standard matrix multiplication definition:

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

A few notes:

- The definition gives a natural \( O(n^3) \) algorithm.
- \( \Omega(n^2) \) lower bound is natural because the size of the output is \( \Theta(n^2) \).
- A widely held belief was that \( \Theta(n^3) \) was the right complexity for matrix multiplication until Strassan “shocked the computing world.”
Strassan’s Amazing Algorithm

We break up the \( n \times n \) matrices into 4 blocks of \( n/2 \). You can assume here that \( n \) is always a power of 2.

\[
X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}
\]

Now the product \( X \) and \( Y \) can be written in terms of the \( n/2 \) blocks:

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

Let’s write down the running time of this algorithm with a recurrence relation. Let \( T(n) \) be the number of scalar multiplications performed on two \( n \times n \) matrices. Then we can write \( T(n) \) as

\[
T(n) = 8T(n/2) + O(n^2)
\]

Where the first term corresponds to the 8 matrix multiplications we perform and the final term corresponds to the matrix additions. Using the master method, this yields an \( O(n^3) \)-time solution. Note that \( a = 8, b = 2 \) and \( d = 2 \) which means that \( a/b^d = 2 \) and \( n^\log_{b^d} a = n^3 \). However, we can pull a trick similar to integer multiplication:

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

Now we can write \( Z \) as

\[
Z = X \times Y = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_3 - P_5 - P_7 \end{bmatrix}
\]

Now we are only performing 7 matrix multiplications, each of size \( n/2 \) so we have:

\[
T(n) = 7T(n/2) + O(n^2) = n^{\log_2 7} \equiv O(n^{2.81})
\]

Some notes:

- Until very recently, the best current algorithm, which is due to Coppersmith and Winograd, yields \( O(n^{2.376}) \). Stothers and Williams recently improved this slightly.
- Strassan’s algorithm has poor numerical properties, so it is often not used.
- Also, the constant factor is much higher for Strassan’s algorithm due to the increased number of additions. Thus, it is typically only reasonable to use this on matrices with dimension \( n > 100 \).