Design and Analysis of Algorithms

Part 2

Divide and Conquer Algorithms

Elias Koutsoupias
with thanks to Giulio Chiribella

Hilary Term 2022

Basic algorithmic techniques

When faced with a new algorithmic problem, one should consider applying one of the following approaches:

- **Divide-and-conquer** :: divide the problem into two subproblems, solve each problem separately and merge the solutions
- **Dynamic programming** :: express the solution of the original problem as a recursion on solutions of similar smaller problems. Then instead of solving only the original problem, solve all sub-problems that can occur when the recursion is unravelled, and combine their solutions
- **Greedy approach** :: build the solution of an optimization problem one piece at a time, optimizing each piece separately
- **Inductive approach** :: express the solution of the original problem based on the solution of the same problem with one fewer item; a special case of dynamic programming and similar to the greedy approach

The divide-and-conquer strategy

The ***divide-and-conquer*** strategy solves a problem by:

1. Breaking it into subproblems (smaller instances of the same problem)
2. Recursively solving these subproblems
   [*Base case*: If the subproblems are small enough, just solve them by brute force.]
3. Appropriately combining their answers.

Where is the work done?

In three places:

1. In dividing the problems into subproblems.
2. At the tail end of the recursion, when the subproblems are so small they are solved outright.
3. In the gluing together of the intermediate answers.

Merge sort [CLRS 2.3.1]

Merge sort is a divide-and-conquer algorithm.

Informal description:

It sorts a subarray $A[p\,..\,r) := A[p\,..\,r-1]$

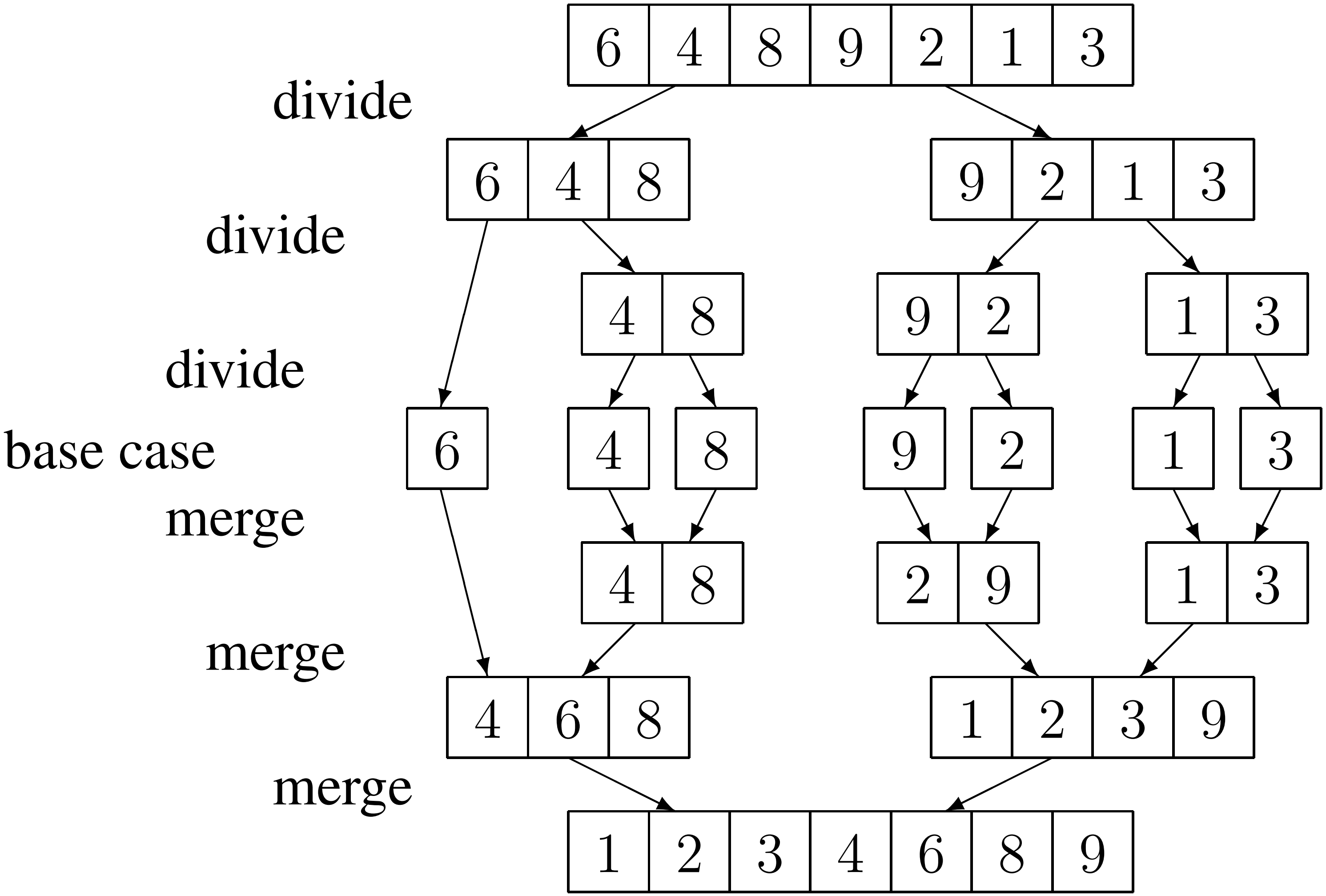
Divide by splitting it into subarrays $A[p\,..\,q)$ and $A[q\,..\,r)$ where $q = \lfloor (p+r)/2 \rfloor$.

Conquer by recursively sorting the subarrays.
Recursion stops when the subarray contains only one element.

Combine by merging the *sorted* subarrays $A[p\,..\,q)$ and $A[q\,..\,r)$ into a single sorted array, using a procedure called MERGE(A, p, q, r).

MERGE compares the two smallest elements of the two subarrays and copies the smaller one into the output array.
This procedure is repeated until all the elements in the two subarrays have been copied.

Example


6 4 8 9 2 1 3
divide
6 4 8
9 2 1 3
divide
4 8
9 2
1 3
divide
base case
6
4
8
9
2
1
3
merge
4 8
2 9
1 3
merge
4 6 8
1 2 3 9
merge
1 2 3 4 6 8 9

Pseudocode for MERGE-SORT

MERGE-SORT(A, p, r)

Input: An integer array A with indices $p < r$.
Output: The subarray $A[p \,..\, r)$ is sorted in non-decreasing order.

if $r > p + 1$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q, r)
    MERGE(A, p, q, r)

Initial call: MERGE-SORT$(A, 1, n + 1)$

Merge

Input: Array A with indices p, q, r such that

- □ $p < q < r$
- □ Subarrays $A[p \mathinner{.\,.} q)$ and $A[q \mathinner{.\,.} r)$ are both sorted.

Output: The two sorted subarrays are merged into a single sorted subarray in $A[p \mathinner{.\,.} r)$.

Pseudocode for MERGE

```
MERGE(A, p, q, r)

n1 = q - p
n2 = r - q
Create array L of size n1 + 1
Create array R of size n2 + 1
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j - 1]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r - 1
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Running time of MERGE

- The first two **for** loops take $\Theta(n_1 + n_2) = \Theta(n)$ time, where $n = r - p$.
- The last **for** loop makes n iterations, each taking constant time, for $\Theta(n)$ time.
- Total time: $\Theta(n)$.

Remark

- The test in line 14 is left-biased, which ensures that MERGE-SORT is a *stable* sorting algorithm: if $A[i] = A[j]$ and $A[i]$ appears before $A[j]$ in the input array, then in the output array the element pointing to $A[i]$ appears to the left of the element pointing to $A[j]$.

Characteristics of merge sort

- The worst-case running time of MERGE-SORT is $\Theta(n \log n)$, much better that the worst-case running time of INSERTION-SORT, which was $\Theta(n^2)$.
  (see next slides for the explicit analysis of MERGE-SORT).
- MERGE-SORT is stable, because MERGE is left-biased.
- MERGE and therefore MERGE-SORT is not in-place:
  it requires $\Theta(n)$ extra space.
- MERGE-SORT is not an online-algorithm: the whole array A must be specified before the algorithm starts running.

Analysing divide-and-conquer algorithms [CLRS 2.3.2]

We often use a ***recurrence*** to express the running time of a divide-and-conquer algorithm.

Let $T(n)$ = running time on a problem of size n.

- If n is small (say $n \leq \ell$), use constant-time brute force solution.
- Otherwise, we divide the problem into a subproblems, each $1/b$ the size of the original.
- Let the time to divide a size-n problem be $D(n)$.
- Let the time to combine solutions (back to that of size n) be $C(n)$.

We get the recurrence

$$T(n) = \begin{cases} c & \text{if } n \leq \ell \\ a\,T(n/b) + D(n) + C(n) & \text{if } n > \ell \end{cases}$$

Example: MERGE-SORT

For simplicity, assume $n = 2^k$.

For $n = 1$, the running time is a constant c.

For $n \geq 2$, the time taken for each step is:

- □ **Divide**: Compute $q = (p + r)/2$; so, $D(n) = \Theta(1)$.
- □ **Conquer**: Recursively solve 2 subproblems, each of size $n/2$; so, $2T(n/2)$.
- □ **Combine**: MERGE two arrays of size n; so, $C(n) = \Theta(n)$.

More precisely, the recurrence for MERGE-SORT is

$$T(n) \; = \; \begin{cases} c & \text{if } n = 1 \\ 2\,T(n/2) + f(n) & \text{if } n > 1 \end{cases}$$

where the function $f(n)$ is bounded as $d'\,n \leq f(n) \leq d\,n$ for suitable constants $d, d' > 0$.

Solving recurrence equations

We will consider three methods for solving recurrence equations:

1. Guess-and-test (called the substitution method in [CLRS])
2. Recursion tree
3. Master Theorem
4. By changing variables

Guess-and-test [CLRS 4.3]

- Guess an expression for the solution. The expression can contain constants that will be determined later.
- Use induction to find the constants and show that the solution works.

Let us apply this method to MERGE-SORT.

The recurrence of MERGE-SORT implies that there exist two constants $c, d > 0$ such that

$$T(n) \leq \begin{cases} c & \text{if } n = 1 \\ 2\,T(n/2) + d\,n & \text{if } n > 1 \end{cases}$$

Guess-and-test [CLRS 4.3]

- Guess an expression for the solution. The expression can contain constants that will be determined later.
- Use induction to find the constants and show that the solution works.

Let us apply this method to MERGE-SORT.

The recurrence of MERGE-SORT implies that there exist two constants $c, d > 0$ such that

$$T(n) \leq \begin{cases} c & \text{if } n = 1 \\ 2\,T(n/2) + d\,n & \text{if } n > 1 \end{cases}$$

Guess. There is some constant $a > 0$ such that $T(n) \leq an \lg n$ for all $n \geq 2$ that are powers of 2.

Let's test it!

Solving the Merge-Sort recurrence by guess-and-test

Test. For $n = 2^k$, by induction on k.

Base case: $k = 1$

$$T(2) = 2c + 2d \leq a\,2\lg 2 \qquad \text{if } a \geq c + d$$

Inductive step: assume $T(n) \leq an\log n$ for $n = 2^k$.
Then, for $n' = 2^{k+1}$ we have:

$$\begin{aligned} T(n') &\leq 2a\tfrac{n'}{2}\lg\left(\tfrac{n'}{2}\right) + d\,n' \\ &= an'\lg n' - an'\lg 2 + d\,n' \\ &\leq an'\lg n' \qquad \text{if } a \geq d \end{aligned}$$

In summary: choosing $a \geq c + d$ ensures $T(n) \leq an\lg n$,
and thus $T(n) = O(n\log n)$.
A similar argument can be used to show that $T(n) = \Omega(n\log n)$.
Hence, $T(n) = \Theta(n\log n)$.

The recursion tree [CLRS 4.4]

Guess-and-test is great, but how do we guess the solution?
One way is to use the ***recursion tree***,
which exposes successive unfoldings of the recurrence.

The idea is well exemplified in the case of MERGE-SORT.
The recurrence is

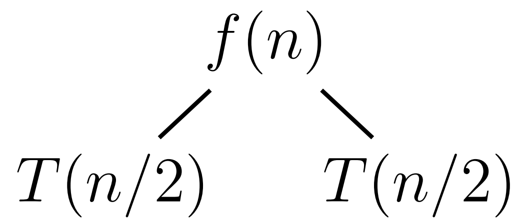
$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2\,T(n/2) + f(n) & \text{if } n > 1 \end{cases}$$

where the function $f(n)$ satisfies the bounds $d'\,n \leq f(n) \leq d\,n$, for suitable constants $d, d' > 0$.

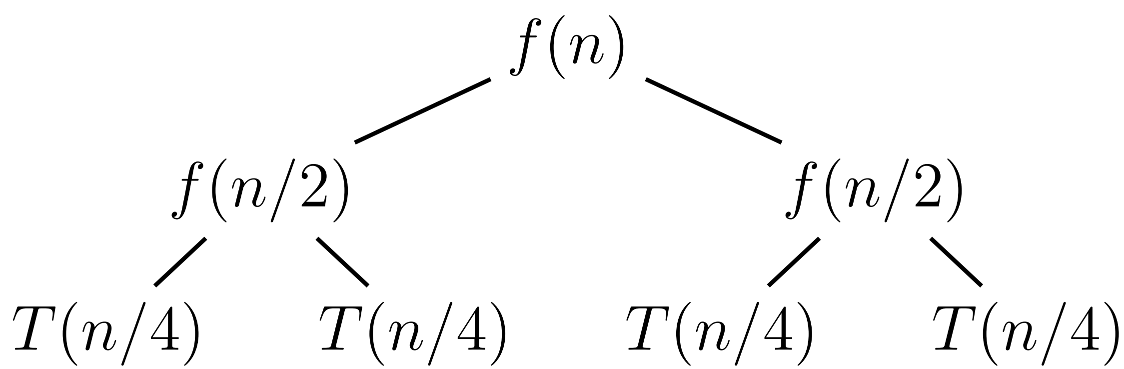
Unfolding the recurrence of MERGE-SORT

Assume $n = 2^k$ for simplicity.

First unfolding: cost of $f(n)$ plus cost of two subproblems of size $n/2$

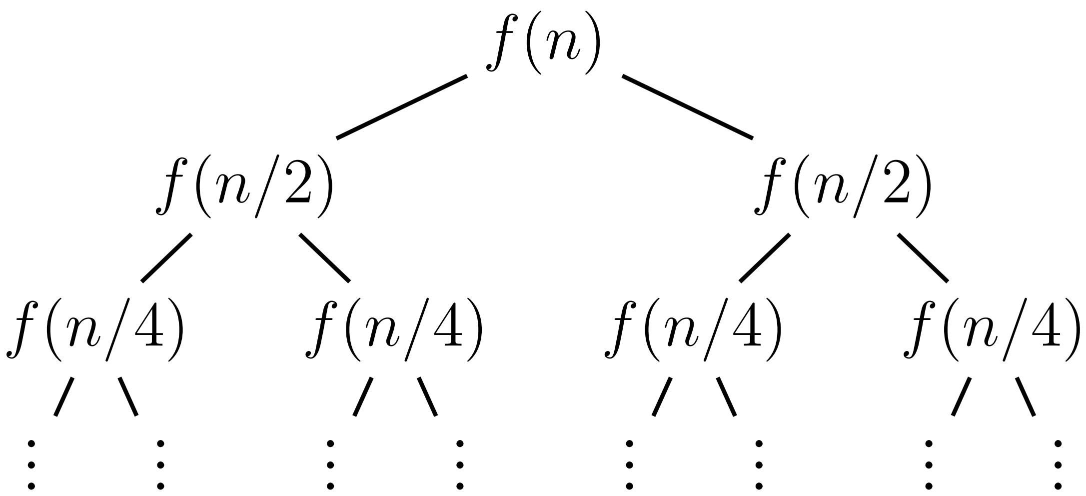


Second unfolding: for each size-$n/2$ subproblem, cost of $f(n/2)$ plus cost of two subproblems of size $n/4$ each.



Unfolding the recurrence of MERGE-SORT (cont'd)

Continue unfolding, until the problem size (= node label) gets down to 1:



In total, there are $\lg n + 1$ levels.

- Level 0 (root) has cost $C_0(n) = f(n)$.
- Level 1 has cost $C_1(n) = 2f(n/2)$.
- Level 2 has cost $C_2(n) = 4f(n/4)$.
- For $l < \lg n$, level l has cost $C_l(n) = 2^l f(n/2^l)$.
  Note that, since $d'\,n \leq f(n) \leq d\,n$, we have $d'\,n \leq C_l(n) \leq d\,n$.
- The last level (consisting of n leaves) has cost cn.

Analysing MERGE-SORT with the recursion tree

The total cost of the algorithm is the sum of the costs of all levels:

$$T(n) = \sum_{l=0}^{\lg n - 1} C_l(n) + c\,n\,.$$

Using the relation $d'\,n \le C_l(n) \le dn$ for $l < \lg n$, we obtain the bounds

$$d'\,n \lg n + c\,n \le T(n) \le d\,n \lg n + c\,n\,.$$

Hence, $T(n) = \Theta(n \log n)$.

The Master Theorem [DPV 2.2]

Theorem. *Suppose*

$$T(n) \leq aT(\lceil n/b \rceil) + O(n^d)$$

for some constants $a > 0$ *and* $b > 1$ *and* $d \geq 0$.
Then,

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

Example: For MERGE-SORT, $a = b = 2$ and $d = 1$.
The master theorem gives $T(n) = O(n \log n)$.

Note. See [CLRS 4.5] for a stronger version of the Master Theorem.

Proof of the Master Theorem*

By a recursion tree argument.

First assume n is a power of b. (We shall relax this later.)

The size of the subproblems decreases by a factor of b at each recursion, and reaches the base case after $\log_b n$ divisions.

Since the branching factor is a, level k of the tree comprises a^k subproblems, each of size n/b^k.

Proof cont'd

The cost at level l is upper bounded by $c\,a^l \times (\frac{n}{b^l})^d = c\,n^d \times (\frac{a}{b^d})^l$, for a suitable constant $c > 0$.
Thus, the total cost is upper bounded by

$$T(n) \;\leq\; c\,n^d \left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \cdots + \left(\frac{a}{b^d}\right)^{\log_b n}\right)\,.$$

Proof cont'd

The cost at level l is upper bounded by $c\,a^l \times (\frac{n}{b^l})^d = c\,n^d \times (\frac{a}{b^d})^l$, for a suitable constant $c > 0$.
Thus, the total cost is upper bounded by

$$T(n) \ \leq \ c\,n^d \left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \cdots + \left(\frac{a}{b^d}\right)^{\log_b n}\right) .$$

Now, there are three cases:

1. $a < b^d$, i.e. $d > \log_b a$: the geometric series sums up to a constant.
   Hence, $T(n) = O(n^d)$.
2. $a = b^d$, i.e. $d = \log_b a$: the geometric series sums up to $1 + \log_b n$.
   Hence, $T(n) = O(n^d \log n)$.
3. $a > b^d$, i.e. $d < \log_b a$: the geometric series sums up to $\Theta\left(\left(\frac{a}{b^d}\right)^{\log_b n}\right)$.
   Since $\left(\frac{a}{b^d}\right)^{\log_b n} = \frac{n^{\log_b a}}{n^d}$, we have
   $T(n) \leq c\,n^d\,\Theta\left(\frac{n^{\log_b a}}{n^d}\right) = \Theta(n^{\log_b a})$. Hence, $T(n) = O(n^{\log_b a})$.

Extension to arbitrary integers

We proved the Master Theorem when n is a power of b.
What about arbitrary n?

Idea: Assume that $T(n)$ is a non-decreasing function of n (as we expect for the running time of an algorithm).
Then, $T(n) \leq T(n')$, where $n' = b^{\lceil \log_b n \rceil}$ is the smallest power of b that is larger than n.

Example: case 2.
We know that $T(n') \leq c\,(n')^d$ for some constant $c > 0$. Then,

$$T(n) \leq T(n') \leq c\,(n')^d \leq c\,b^{d\lceil \log_b n \rceil} \leq c\,b^{d(\log_b n + 1)} \leq c'\,n^d\,,$$

with $c' = c\,b^d$. Hence, $T(n) = O(n^d)$.

The same reasoning applies to cases 2 and 3.

Changing variables

Consider the recurrence

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

which, at first sight, does not fit the form of the Master Theorem.

A trick. By introducing the variable $k = \log n$ we get

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

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

$$S(k) \;=\; 2S(k/2) + k$$

By the Master Theorem, we have $S(k) = O(k \log k)$, and so

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

Further examples of divide-and-conquer algorithms

In the following, we will see divide-and-conquer algorithms for

- integer multiplication
- matrix multiplication
- search (in a sorted array)
- selection (finding the i-th smallest element in an array)
- Fast Fourier Transform

Example 1: Integer Multiplication [DPV 2.1]

An old observation of Carl Gauss (1777-1855)

Product of complex numbers

$$(a + ib)(c + di) = ac - bd + (bc + ad)i$$

can be done with just three real-number multiplications

$$ac, \quad bd, \quad (a + b)(c + d)$$

because $bc + ad = (a + b)(c + d) - ac - bd.$

Can we exploit Gauss' trick for the multiplication of binary integers?

Multiplying n-bit integers

Divide and conquer: Split each of n-bit numbers x and y into their left and right halves, which are each $n/2$-bits long:

$$x = \boxed{x_L}\boxed{x_R} = 2^{n/2}x_L + x_R$$

$$y = \boxed{y_L}\boxed{y_R} = 2^{n/2}y_L + y_R$$

Since

$$\begin{aligned} xy &= (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) \\ &= 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

compute xy by four $(n/2)$-bit multiplications $x_L y_L, x_L y_R, x_R y_L, x_R y_R$, three additions and two multiplications by powers of 2 (= left-shifts).

Writing $T(n)$ for run time on multiplying n-bit inputs, we have

$$T(n) = 4T(n/2) + O(n), \text{ and so } T(n) = O(n^2).$$

A faster multiplication (Karatsuba and Ofman)

Using Gauss' trick, **three** $(n/2)$-bit multiplications suffice:

$$x_L y_L, \quad x_R y_R, \quad (x_L + x_R)(y_L + y_R).$$

Reducing the number of multiplications from 4 to 3 may not look impressive, but this little saving ***occurs at every level of the recursion***.

Thanks to it, the running time is $T(n) = 3T(n/2) + O(n)$, and the Master Theorem yields

$$T(n) \;=\; O(n^{\log_2 3}) \;\approx\; O(n^{1.59})$$

A significant improvement!

Example 2: Matrix multiplication [DPV 2.5, CLRS 4.2]

Let X be a $p \times q$ matrix and Y be a $q \times r$ matrix. The product $Z = X \cdot Y$ is a $p \times r$ matrix where

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

Standard algorithm. The above definition yields an algorithm requiring $p \times q \times r$ multiplications and $p \times (q-1) \times r$ additions. In case $p = q = r = n$, the total cost is $2n^3 - n^2 = O(n^3)$ operations.

Can we do better?

Strassen's divide-and-conquer method (1969)

View X and Y as each composed of four $n/2 \times n/2$ blocks:

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

Strassen's method

Then XY can be expressed in terms of these blocks (which behave as if they are singletons):

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

We use a divide-and-conquer strategy. To compute size-n product XY, recursively compute eight size-$(n/2)$ products:

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

then do some $O(n^2)$-time additions.

Running time: $T(n) = 8T(n/2) + O(n^2)$, which gives $T(n) = O(n^3)$, thanks to the Master Theorem.

This is unimpressive.

Strassen's trick*

Size-n XY can be computed from just ***seven*** size-$(n/2)$ subproblems.

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

where

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

The new running time is $T(n) = 7T(n/2) + O(n^2)$; hence

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81}).$$

Example 3: Search [CLRS Exercise 2.3-5]

The Search Problem:

Input: A subarray $A[p, \ldots, r)$ of distinct integers sorted in increasing order, and an integer z

Output: “Yes” if z appears in $A[p, \ldots, r)$, “No” otherwise.

BINSEARCH(A, p, r, z)

// Assume A sorted in increasing order
if $p \geq r$
    return “No”
else $q = \lfloor (p + r)/2 \rfloor$
    if $z = A[q]$
        return “Yes”
    else if $z < A[q]$
        BINSEARCH(A, p, q, z)
    else BINSEARCH$(A, q + 1, r, z)$

Running time of BINSEARCH

Let $T(n)$ be the worst-case running time of BINSEARCH on an input array of length $n = r - p$. Then

$$T(n) \; \leq \; \begin{cases} O(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + O(1) & \text{otherwise} \end{cases}$$

By the Master Theorem, $T(n) = O(\log n)$.

Example 4: Selection [CLRS 9.3]

The ***ith-order statistic*** of a set of n (distinct) elements is the i-th smallest element (i.e. the element that is larger than exactly $i - 1$ other elements).

The ***median*** is the $\lfloor (n+1)/2 \rfloor$-order statistics.

The Selection Problem:

Input: A set of n (distinct) numbers and a number i, with $1 \leq i \leq n$.

Output: The ith-order statistic of the set.

An upper bound

The selection problem can be solved in $O(n \log n)$ time:

- □ Sort the numbers in $O(n \log n)$ time using MERGE-SORT
- □ Return the i-th element in the sorted array.

But do we really need to sort first? Can't we find a faster algorithm?

A fast algorithm for selection

Using a divide-and-conquer approach, one can find the i-th smallest element in $O(n)$ time, even *in the worst case*!

The algorithm SELECT is based on two ideas:

Idea 1: pick an element of the array $A[1\,..\,n]$, say $A[q]$, called the *pivot*. Partition the array into three subarrays, one containing the elements smaller than $A[q]$, one containing $A[q]$, and one containing the elements larger than $A[q]$.
Reduce the search for the i-th element to one of the subarrays.

Idea 2: Choose the element $A[q]$ in such a way that the subarray of elements larger than $A[q]$ and the subarray of elements smaller than $A[q]$ are of comparable size.

A fast algorithm for selection

Using a divide-and-conquer approach, one can find the i-th smallest element in $O(n)$ time, even *in the worst case*!

The algorithm SELECT is based on two ideas:

Idea 1: pick an element of the array $A[1\,..\,n]$, say $A[q]$, called the *pivot*.
Partition the array into three subarrays, one containing the elements smaller than $A[q]$, one containing $A[q]$, and one containing the elements larger than $A[q]$.
Reduce the search for the i-th element to one of the subarrays.

Idea 2: Choose the element $A[q]$ in such a way that the subarray of elements larger than $A[q]$ and the subarray of elements smaller than $A[q]$ are of comparable size.
To do so, divide the array A into small groups (e.g. of size 5 or less), find the median of each group, and compute the median of the medians.
Choose $A[q]$ to be the median of medians.

(Of course, to find the median of medians we need to run SELECT. But the point is that the size of the input has been reduced from n to $\lceil n/5 \rceil$.)

The partition task

Input: An input subarray $A[p\,..\,r]$, containing distinct numbers, and an array element $A[q]$ (the *pivot*)

Output: An output subarray $A'[p\,..\,r]$ and an array index q' such that

- ☐ $A'[p\,..\,r]$ consists of the same *set* of numbers as $A[p\,..\,r]$
- ☐ $A'[p\,..\,q'-1]$ consists of numbers $< A[q]$
- ☐ $A'[q'] = A[q]$.
- ☐ $A'[q'+1\,..\,r]$ consists of numbers $> A[q]$.

It is easy to see that the partition task can be implemented in $O(n)$ time, with $n = r - p + 1$.
One has only to go through the elements of A, and to copy the element $A[i]$ ($i \neq q$) into one of two arrays, B and C, depending on whether $A[i] < A[q]$ or $A[i] > A[q]$. Then, the two arrays B and C can be used to build an array A' with the desired properties.

More interestingly, the partition can be done *in place*, see CLRS 7.1 for an explicit algorithm.

The algorithm $\textsc{Select}(A, i)$

Input: An array A of n *distinct* numbers.

Output: The i-th smallest element.

1. Divide the n input elements into $\lfloor n/5 \rfloor$ groups of 5 elements each, and at most one group of the remaining $n \bmod 5$ elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups (e.g. by running INSERTION-SORT and picking the appropriate element)
3. Use SELECT to find the ***median-of-medians***, call it x.
4. Use x as pivot, to partition the input array into three subarray.
5. Compute the number of elements in the lower subarray (consisting of elements $< x$), and denote it by k.
6. Three cases:
   (a) If $i = k + 1$, return x.
   (b) If $i < k + 1$, call SELECT to find i-th element of the lower subarray.
   (c) If $i > k + 1$, call SELECT to find $(i - k - 1)$-th element of the upper subarray.

Running time analysis of SELECT

Let $T(n)$ be running time of SELECT on an array of n elements.
By definition $T(n) = \sum_j T_j(n)$, where $T_j(n)$ is the cost of implementing line j of the program.

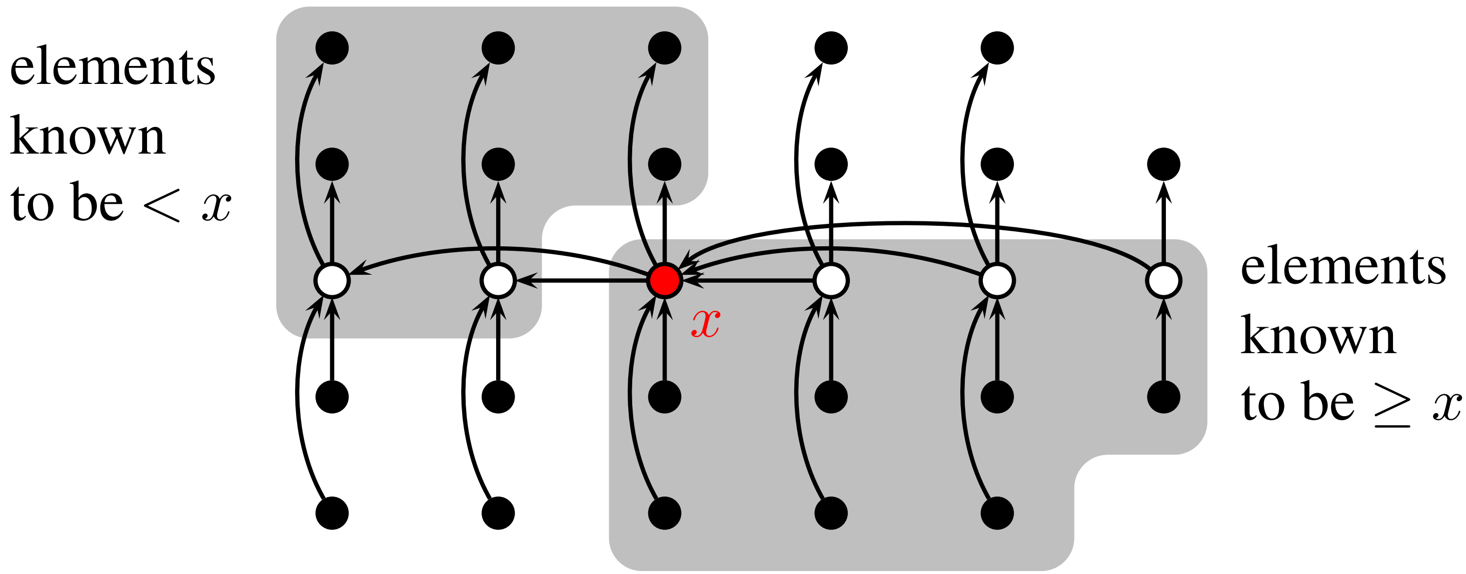
1. Line 1 (dividing the input array) costs $O(n)$ time
2. Line 2 (computing $\lceil n/5 \rceil$ "baby medians") costs $O(n)$
3. Line 3 (finding the median of medians) costs $T(\lceil n/5 \rceil)$
4. Line 4 (partitioning) costs $O(n)$
5. Line 5 (computing size of subarrays) costs $O(1)$
6. Line 6 (selecting within a subarray) costs at most $T(|S_{\max}|)$, where $|S_{\max}|$ is the size of the largest subarray.

Assuming that $T(n)$ is non-decreasing, we have the recurrence

$$T(n) \leq T(\lceil n/5 \rceil) + T(|S_{\max}|) + O(n)$$

Bounding the size of the subarrays

By definition, at least half of the $\lceil n/5 \rceil$ groups have "baby medians" $\geq x$. Each of these groups has at least 3 elements $> x$, except for the group containing x and, possibly, for the group with fewer than 5 elements.



Thus the number of elements $> x$ is at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

Hence, the size of the lower subarray (elements $< x$) is upper bounded by $7n/10 + 6$.

Bounding the size of the subarrays (cont'd)

A similar argument applies to the upper subarray:

- □ At least half of the $\lceil n/5 \rceil$ groups have "baby medians" $\leq x$.
- □ Each of those groups has at least 3 elements $< x$, except for the group containing x and, possibly, for the group with fewer than 5 elements.
- □ The number of elements $< x$ is at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

- □ The size of the upper subarray (elements $> x$) is upper bounded by $7n/10 + 6$.

Since the size of each subarray is an integer, we have the bound $|S_{\max}| \leq \lfloor 7n/10 + 6 \rfloor$.

Solving the recurrence of SELECT by guess-and-test

Assuming that $T(n)$ is non-decreasing, we have the recurrence

$$T(n) \le T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + b\,n$$

for some constant $b > 0$.

Guess. *There is some* $c > 0$ *such that* $T(n) \le c\,n$ *for all* $n > 0$.

A useful observation

Substituting the guess into the recurrence, we get

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c\lfloor 7n/10 + 6 \rfloor + bn \\ &\leq cn/5 + c + 7cn/10 + 6c + bn \\ &= 9cn/10 + 7c + bn \\ &= cn + (-cn/10 + 7c + bn) \end{aligned}$$

which is at most cn provided that $-cn/10 + 7c + bn \leq 0$ or, equivalently,

$$c \geq 10bn/(n-70).$$

Now, if $n \geq 140$, we have $n/(n-70) \leq 2$.
Hence, the inequality is satisfied if $n \geq 140$ and $c \geq 20b$.

Validity of the guess

Lemma. *There is some $c > 0$ such that $T(n) \leq c\,n$ for all $n > 0$.*

Proof.
Let $a = \max\{T(n)/n\,,\ n \leq 140\}$.
Define $c = \max\{a, 20b\}$.

Base case: For every $n \leq 140$, $T(n) \leq c\,n$ by construction.

Inductive case: Suppose that the condition $T(n) \leq c\,n$ holds for all n up to $n_0 \geq 140$. Then, for $n = n_0 + 1$ we have

$$
\begin{aligned}
T(n) &\leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + b\,n \\
&\leq c\lceil n/5 \rceil + c\lfloor 7n/10 + 6) \rfloor + b\,n \\
&\leq cn\,,
\end{aligned}
$$

by construction (see previous slide).

Epilogue: selection vs sorting

- SELECT finds the i-th smallest element in $O(n)$ time.
- our best sorting algorithm so far, MERGE-SORT, sorts the array in $O(n \log n)$ time.

It seems that finding the i-th smallest element of an array is much easier than sorting the whole array.

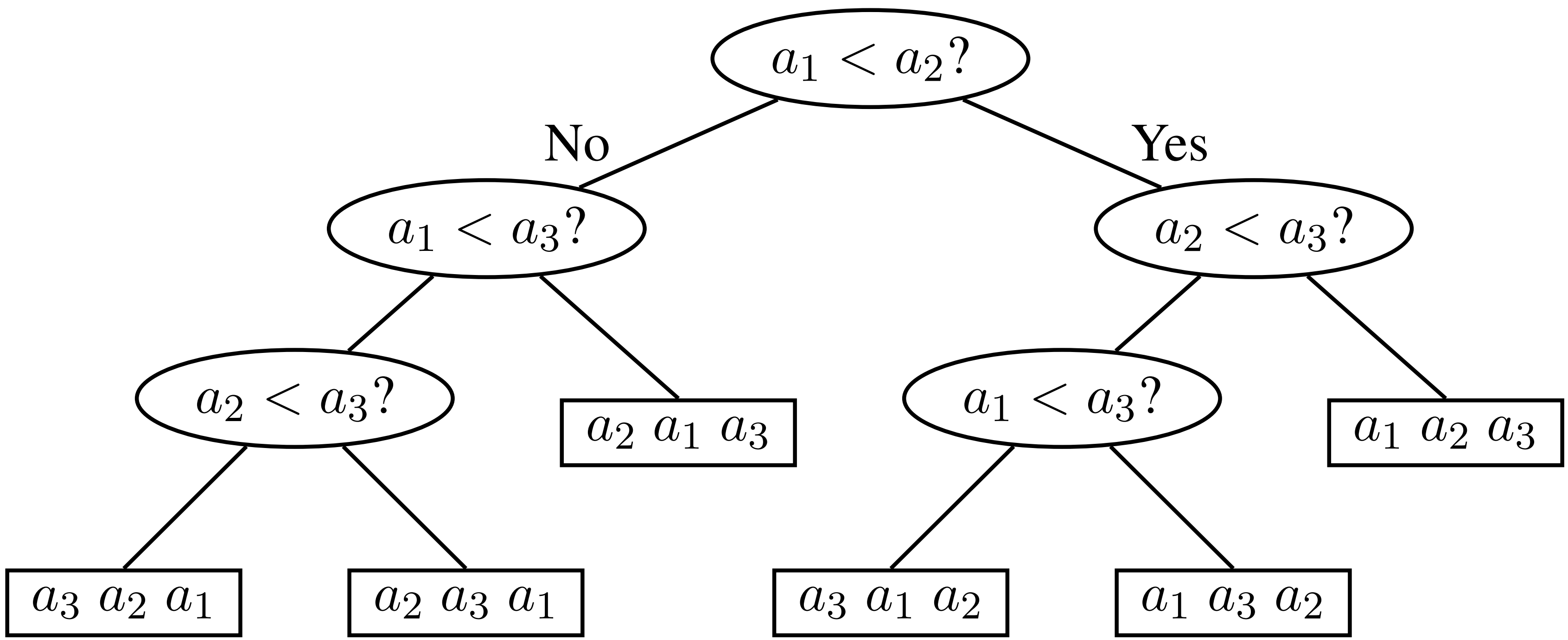
Is this true?

- Yes, if the sorting algorithm is based on comparisons between elements of the array
- No, if we know that the entries of the input array are contained in an interval of size $k = O(n)$. In that case, there exists a sorting algorithm that runs in $O(n)$ time.

A lower bound for comparison-based sorting [CLRS 8.1]

Theorem 1. *The running time of every* comparison-based *sorting algorithm is* $\Omega(n \log n)$.

Proof. Consider the ***decision tree*** of a comparison-based algorithm on input sequence $a_1\ a_2\ a_3$:



Observation. The *depth* of the tree (= number of comparisons on the longest branch) is the worst-case time complexity of the algorithm.

A lower bound for sorting, cont'd.

Aim. *Obtain a lower bound on the depth of a decision tree.*

The decision tree has $n!$ leaves.

- □ By construction every leaf is labelled by a permutation of $\{ a_1, a_2, \cdots, a_n \}$.
- □ Every permutation must appear as the label of a leaf.
  (Why? Because every permutation could be a valid output)
  Hence the decision tree has at least $n!$ leaves.

Fact. *Every binary tree of depth* d *has at most* 2^d *leaves*
(Proof. Easy induction on d.)

Thus the depth of the decision tree — and the worst-case complexity of the algorithm — is at least $\log(n!)$.

Finally note that $\log(n!) = \Omega(n \log n)$ (Exercise).

Sorting without comparisons [CLRS 8.2]

Example: Counting sort

- □ Based, not on comparison, but on the assumption that each of the n input elements is an integer in the range 0 to k.
- □ Counting sort determines for each input element x the number of elements less than x.
- □ If m elements are less than x, then x belongs in $(m+1)$-th position.
- □ This scheme has to be modified slightly to handle multiple elements with the same value (see line 12 in the following pseudocode).
- □ When $k = O(n)$ the algorithm runs in $\Theta(n)$ time.

CountingSort

CountingSort(A, k)

Input: An array $A[1\,..\,n]$ of elements with keys $a_i \in \{0, \ldots, k\}$.
Output: An array B consisting of a sorted permutation of A

Create array C of size $k + 1$
for $i = 0$ **to** k
    $C[i] = 0$
for $j = 1$ **to** n
    $C[A[j]] = C[A[j]] + 1$
// $C[i]$ now contains the number of elements **equal to** i.
for $i = 1$ to k
    $C[i] = C[i] + C[i-1]$
// $C[i]$ now contains the number of elements **less than or equal to** i.
for $j = n$ **downto** 1
    $B[C[A[j]]] = A[j]$
    $C[A[j]] = C[A[j]] - 1$

Example

Input A

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

At line 6 C

0	1	2	3	4	5
2	0	2	3	0	1

At line 9 C

0	1	2	3	4	5
2	2	4	7	7	8

line 12 (1st time) B

1	2	3	4	5	6	7	8
						3	

C

0	1	2	3	4	5
2	2	4	6	7	8

line 12 (2nd time) B

1	2	3	4	5	6	7	8
	0					3	

C

0	1	2	3	4	5
1	2	4	6	7	8

line 12 (3rd time) B

1	2	3	4	5	6	7	8
	0				3	3	

C

0	1	2	3	4	5
1	2	4	5	7	8

line 12 (last time) B

1	2	3	4	5	6	7	8
0	0	2	2	3	3	3	5

Analysis of COUNTINGSORT

- The first and third **for** -loops take $\Theta(k)$ time, where $\{0\,..\,k\}$ is the range the keys are drawn from.
- The second and fourth **for** -loops take $\Theta(n)$ time, where n is the size of the input array.
- Hence the overall time is $\Theta(n+k)$. If $k = O(n)$ then the overall time is $\Theta(n)$.
- In the last **for** -loop the elements of A are taken from right to left to make this sorting algorithm *stable*.

The Fast Fourier Transform (FFT)* [CLRS 30]

FFT is a fundamental algorithm that is based on a divide-and-conquer approach and can be used to multiply polynomials in $\Theta(n \log n)$ arithmetic operations.

Multiplication of polynomials

Given two polynomials of degree $n-1$

$$\begin{aligned} A(x) &= a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} \\ B(x) &= b_0 + b_1 x + \cdots + b_{n-1} x^{n-1} \end{aligned}$$

compute their product (of degree $2n-2$)

$$A(x) \cdot B(x) = c_0 + c_1 x + \cdots c_{2n-2} x^{2n-2},$$

where

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}.$$

Question: can you reduce integer multiplication to polynomial multiplication? what is an appropriate value for x?

Two representations of polynomials

We can represent a polynomial $A(x)$ in two ways

- □ the coefficient form $A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$
- □ the point-value form $(A(v_0), \ldots, A(v_{n-1}))$, for some fixed values $v_0, \ldots, v_{n-1}$.

The first form is good for evaluating a polynomial at a given point: ***Horner's rule*** takes $O(n)$ arithmetic operations

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0 a_{n-1}) \cdots))$$

The second form is good for multiplying polynomials:

$$(A(v_0)B(v_0), \ldots, A(v_{n-1})B(v_{n-1})),$$

takes $O(n)$ arithmetic operations.

Converting representations

- ***Evaluation:*** coefficient form $\Rightarrow$ point-value form
  - Evaluate $A(x)$ at $x_0, x_1, \ldots, x_{n-1}$. Cost $O(n^2)$ using Horner's rule
- ***Interpolation:*** point-value form $\Rightarrow$ coefficient form
  - Given $\{(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})\}$ solve following linear system

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Vandermonde matrix

Matrix $V(x_0, x_1, \ldots, x_{n-1})$ above is called a ***Vandermonde matrix***. It has determinant

$$\prod_{0 \le j < k < n} (x_k - x_j),$$

It is non-singular when the x_i's are distinct.
Thus $A(x)$ is uniquely determined by its point-value representation, and can be found by Gaussian elimination using $O(n^3)$ arithmetic operations
We can directly compute $A(x)$ with $O(n^2)$ arithmetic operations using ***Lagrange's formula:***

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k}(x - x_j)}{\prod_{j \neq k}(x_k - x_j)}$$

(Easy to directly verify that this satisfies $A(x_i) = y_i$.)

Can we do better?

- To multiply two polynomials $A(x)$ and $B(x)$ in coefficient form:
  - Convert to point-value form (evaluation)
  - Multiply in point-value form (linear number of operations)
  - Convert back to coefficient form (interpolation)
- This takes $\Theta(n^2)$ arithmetic operations.
- ***We can improve this by a clever choice of evaluation points***
- **Theorem.** The product of two polynomials of degree-bound n can be computed with $O(n \log n)$ arithmetic operations, with both input and output representations in coefficient form.

Understanding the key idea

Let's focus on speeding up the evaluation of a polynomial: Given a polynomial

$$A(X) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$$

in coefficient form, ***select appropriate points*** $v_0, \ldots, v_{n-1}$ ***to compute*** $A(v_0), \ldots, A(v_{n-1})$ ***as fast as possible***.

Assume that n is even (or even better a power of 2) and observe that

$$A(x) = B(x^2) + x\, C(x^2),$$

where B and C have degree bound $n/2$:

$$B(z) = a_0 + a_2 z + \ldots + a_{n-2} z^{n/2-1}$$
$$C(z) = a_1 + a_3 z + \ldots + a_{n-1} z^{n/2-1}$$

Understanding the key idea (cont.)

We can save a lot, if we evaluate $A(x)$ at pairs of opposite points x and $-x$ (their computation differs in a single arithmetic operation):

$$A(x) = B(x^2) + x\,C(x^2)$$
$$A(-x) = B(x^2) - x\,C(x^2)$$

Let's ***apply this idea recursively*** to $B(x^2)$ and $C(x^2)$.
But wait! To apply the same trick to B (and C), we need to select points to get both $B(x^2)$ and $B(-x^2)$? This seems impossible, unless we use complex numbers!

- top level suggest to use points of the form : x and $-x$
- next level suggest to use points of the form : x, $-x$, ix, $-ix$.
- $\ldots$
- for all levels use points $\omega_n^0, \ldots, \omega_n^{n-1}$, where $\omega_n = e^{2\pi i/n}$ is a primitive n-th root of unity (here we further simplify by setting $x = 1$).

Pseudocode for recursive FFT

RECURSIVE-FFT$(a[0\,..\,n-1])$

if $n = 1$
    return a
$u =$ RECURSIVE-FFT$(a[0, 2, \ldots, n-2])$
$v =$ RECURSIVE-FFT$(a[1, 3, \ldots, n-1])$

$\omega_n = e^{2\pi i/n}$; $x = 1$ // Store powers of ω_n in x
for $k = 0$ **to** $n/2 - 1$ // invariant $x = \omega_n^k$
    $y_k = u_k + x \cdot v_k$
    $y_{k+n/2} = u_k - x \cdot v_k$
    $x = x \cdot \omega_n$
return y

The algorithm performs $O(n \log n)$ arithmetic operations.

Interpolation

What about the opposite process of getting the coefficients $a = (a_0, \ldots, a_{n-1})$ of A from its values $y = (A(\omega^0), \ldots, A(\omega^{n-1}))$?

It turns out that ***the same algorithm works***! Why?

FFT computes

$$y = \mathcal{V}(\omega)a,$$

where

$$\mathcal{V}(\omega) = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \ldots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)(n-1)} \end{pmatrix}$$

is the Vandermonde matrix of the n-th roots of unity. Interpolation asks to compute

$$a = \mathcal{V}(\omega)^{-1}y.$$

The important property is that $\mathcal{V}(\omega)^{-1} = \mathcal{V}(\omega^{-1})/n$.