UNSW SYDNEY

Module 2: Divide and Conquer

Raveen de Silva (K17 202)
Course Admins: cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
UNSW Sydney

Term 1, 2024

Module Learning Outcomes

- Explain how divide-and-conquer techniques are used to develop algorithms
- Solve problems by creatively applying divide-and-conquer techniques
- Communicate algorithmic ideas at different abstraction levels
- Evaluate the efficiency of algorithms and justify their correctness
- Apply the LaTeX typesetting system to produce high-quality technical documents

Table of Contents 3

Table of Contents

Problem

We are given 27 coins of the same denomination; we know that one of them is counterfeit and that it is lighter than the others. Find the counterfeit coin by weighing coins on a pan balance only three times.

Hint

You can reduce the search space by a third in one weighing!

Solution

- Divide the coins into three groups of nine, say A, B and C.
- Weigh group A against group B.
  - If one group is lighter than the other, it contains the counterfeit coin.
  - If instead both groups have equal weight, then group C contains the counterfeit coin!
- Repeat with three groups of three, then three groups of one.

- This method is called “divide-and-conquer”.
- We have already seen some prototypical “serious” algorithms designed using this method: binary search, merge sort and quicksort.
- We’ll now review these algorithms from a divide-and-conquer perspective, and adapt them to solve problems.

Table of Contents 8

- Steps:
  - **Divide:** Test the midpoint of the search range ($\Theta(1)$)
  - **Conquer:** Search one side of the midpoint recursively
  - **Combine:** Pass the answer up the recursion tree ($\Theta(1)$)
- Recursion is $\log_2 n$ levels deep, with a total of $\Theta(1)$ time spent in each level.
- Time complexity is $\Theta(\log n)$.

- Lower bound: find the smallest index i such that $A[i] \geq x$.
- Upper bound: find the smallest index i such that $A[i] > x$.
- Equal range: find the range of indices $\ell..r$ such that $A[\ell] = \ldots = A[r] = x$.

- Decision problems are of the form
  Given some parameters including X, can you . . .
- Optimisation problems are of the form
  What is the smallest X for which you can . . .
- An optimisation problem is typically *much* harder than the corresponding decision problem, because there are many more choices.
- Can we reduce (some) optimisation problems to decision problems?

Example problem: Maximum Median

Definition

Recall that the median of an array is the *middle value* in sorted order.
For example, the median of $[1, 2, 2, 4, 5]$ is 2.

Problem

Ling has an array A consisting of $2n - 1$ integers, sorted from smallest to largest. Note that the median is initially $A[n]$.

She wants her numbers to grow big and strong, so for each of the following k days she adds 1 to one of her numbers.

At the end of the k days, what is the largest possible median that her array can have?

Example problem: Maximum Median

Example

If the starting array is $[1, 2, 2, 4, 5]$ and $k = 3$, the maximum median is 4, achieved by adding 1 to the third number on the first two days and to any number on the last day.

Using all three increases on the third number does not make the median value 5.

Example problem: Maximum Median

- Why is this problem hard?
- Which numbers should we increase, and how many times?

Observation

We are only concerned with the middle number and everything after it. Numbers from the first half of the array can be left small.

- But we still don't know how much to increase the numbers in the second half!
- What if we did know? What if we had a target to aim for, and we just had to report whether it was possible or not?

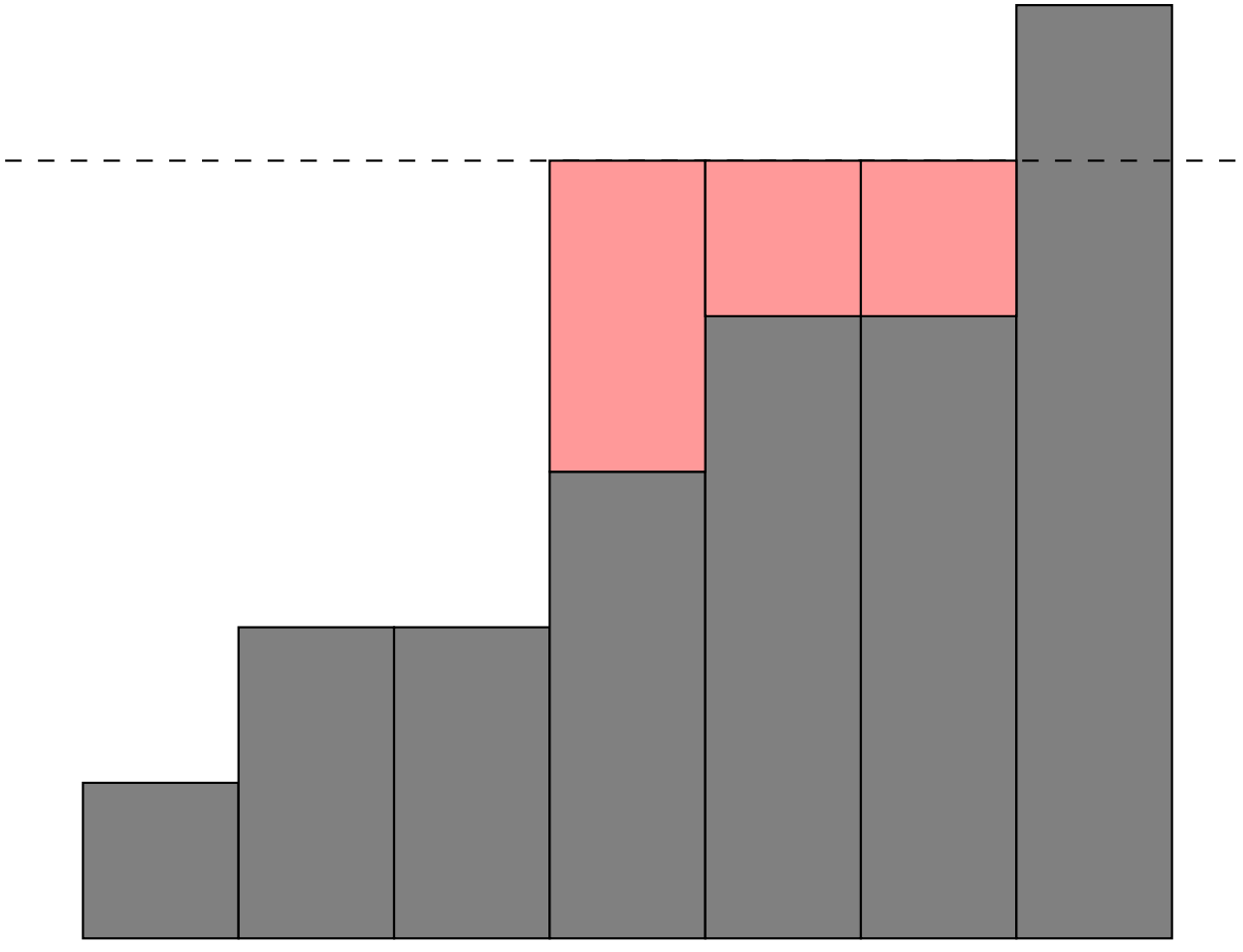
Example problem: Maximum Median

Related problem

Ling wants to know whether she can make the median of her numbers at least some target value t.

- How do we get the median to at least t?
- Increase the middle value $A[n]$ and everything after it to t; nothing to do for any values that are $\geq t$ already.
- Why might we fail?
- t could be too big, that is, we could run out of days.
- Unused days are fine.

Example problem: Maximum Median



Example problem: Maximum Median

- To increase the median to at least t, we will need to increase the values:
  - at index n by $\max(t - A[n], 0)$
  - at index $n+1$ by $\max(t - A[n+1], 0)$
  - ...
  - at index $2n-1$ by $\max(t - A[2n-1], 0)$.

Algorithm (for related problem)

Iterate over indices i from n to $2n-1$ to calculate

$$\sum_{i=n}^{2n-1} \max(t - A[i], 0).$$

If this value is $\leq k$, report yes; otherwise report no.

- This algorithm clearly runs in $O(n)$ in the worst case.

Example problem: Maximum Median

- How is this related to the original problem?
- We can now test whether it's possible to reach (or exceed) any target in $O(n)$ time.
- What are the possible targets?
  - Anything less than the current median $A[n]$ can't possibly be the maximum median.
  - The median can't become any larger than $A[n] + k$, as we don't have enough days to grow $A[n]$ that high, let alone $A[n+1]$ and so on.
- Do we have to try all the targets from $A[n]$ to $A[n] + k$?
- If we test a target t, does the result give us any information about *other* targets?

Example problem: Maximum Median

- If a target t is achievable, then any smaller target is also achievable! Whatever sequence of $+1$s made the median at least t also makes it at least $t-1$, and so on.
- If a target t isn't achievable, then any larger target is also not achievable.
- Therefore targets are all achievable until some cutoff, and all impossible thereafter. Monotonic!
- We can binary search on "is target t achievable?" for t in the range $[A[n], A[n]+k]$.
- We don't have a precomputed array of yes/no values; instead we'll compute them as they're needed in the binary search.
- The binary search has $O(\log k)$ steps each taking $O(n)$, so the time complexity is $O(n \log k)$.

- Suppose we have an optimisation problem involving an integer parameter X.[1]
- Let $f(x)$ be the outcome of the decision problem when $X = x$, with 0 for false and 1 for true.
- In some (but not all) such problems, if the condition holds with $X = x$ then it also holds with $X = x - 1$.
- Thus f is all 1's up to the first 0, after which it is all 0's. So we can use binary search!

[1] Similar ideas work for real X.

- This technique of binary searching the answer, that is, finding the largest x such that $f(x) = 1$ using binary search, is often called *discrete* binary search.

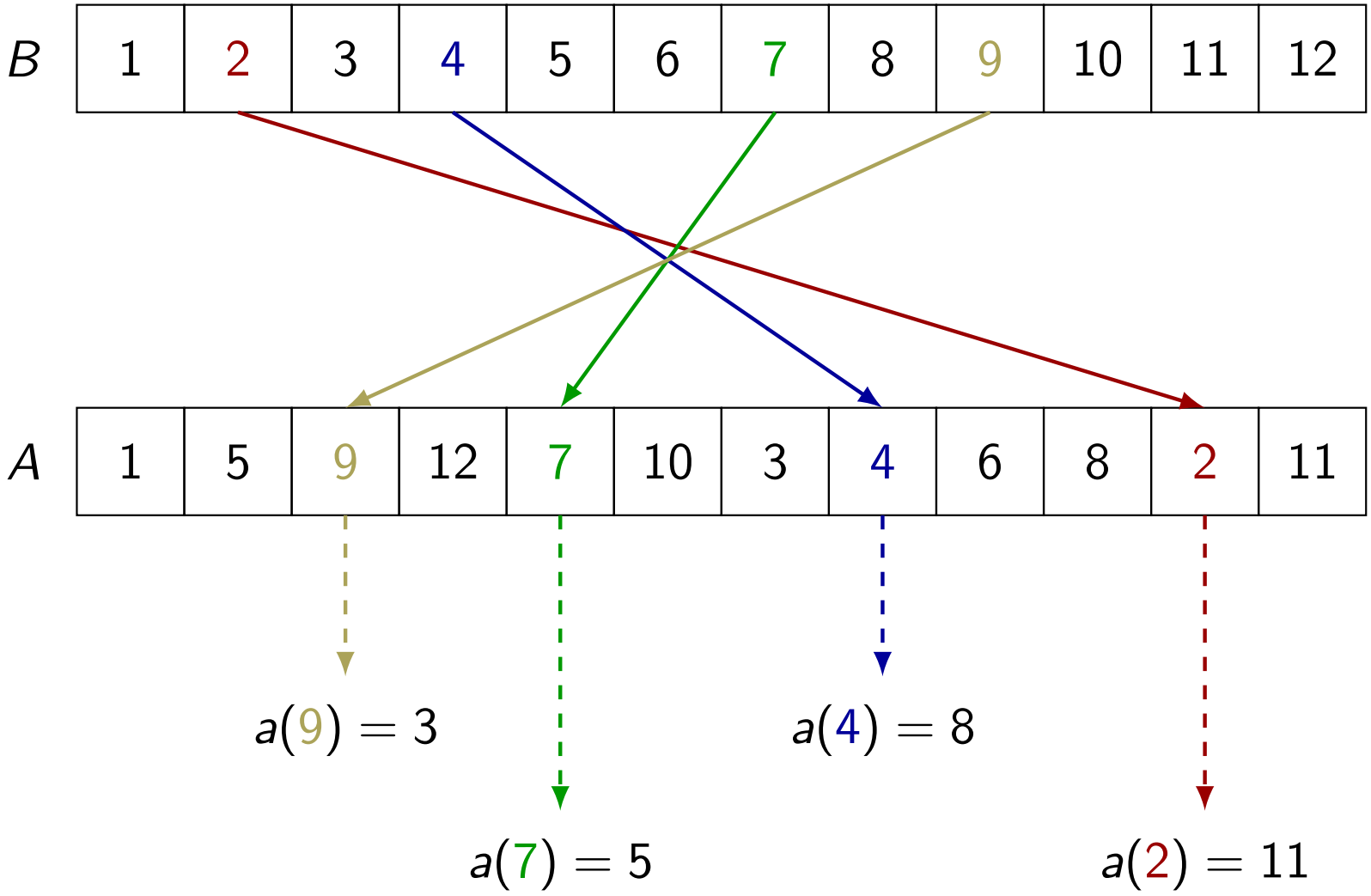
- Overhead is just a factor of $O(\log A)$ where A is the range of possible answers.

Table of Contents

- Steps:
  - **Divide:** Split the array into two equal parts ($\Theta(1)$)
  - **Conquer:** Sort each part recursively
  - **Combine:** Merge the two sorted subarrays ($\Theta(n)$)
- Recursion is $\log_2 n$ levels deep, with a total of $\Theta(n)$ time spent in each level.
- Time complexity is $\Theta(n \log n)$.

- Suppose that you have m users ranking the same set of n movies. You want to determine for any two users A and B how similar their tastes are (for example, in order to make a recommender system).
- How should we measure the degree of similarity of two users A and B?
- Let's number the movies according to B's ranking: assign index 1 to user B's favourite movie, index 2 to their second favourite and so on.
- For each movie i, its rank in A's list will be denoted by $a(i)$.

Counting the number of inversions



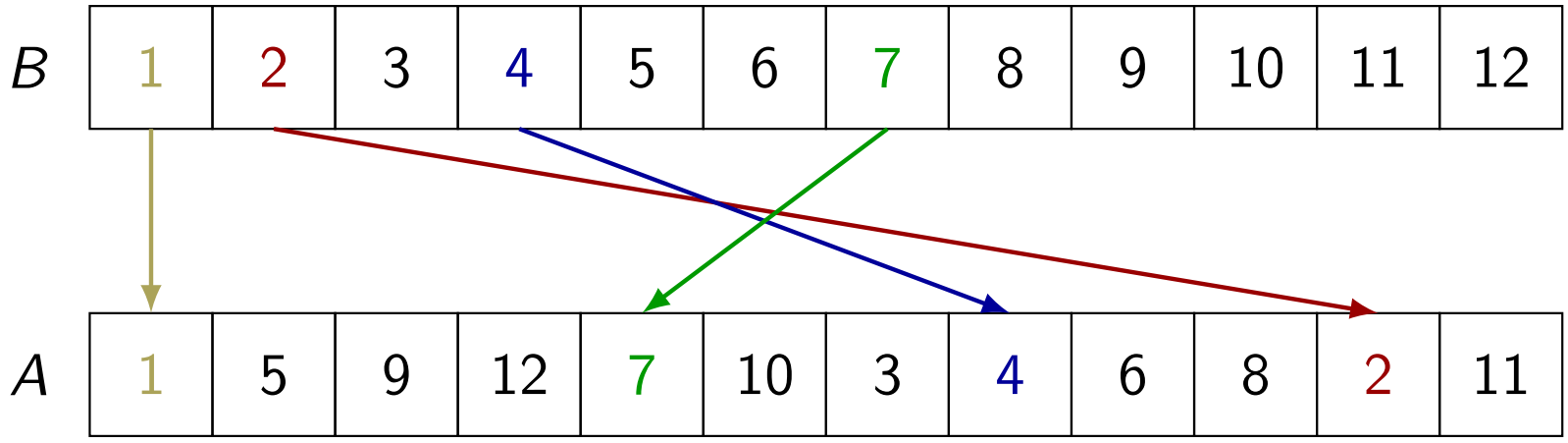
A good measure of how different these two users are is the number of pairs of movies which are 'out of order' between the two lists.

Definition

An *inversion* is a pair (i, j) such that:

- $i < j$, i.e. B prefers i to j, and
- $a(i) > a(j)$, i.e. A prefers j to i.

Our task will be to count the total number of inversions.



For example, movies 1 and 2 do not form an inversion because both users prefer movie 1:

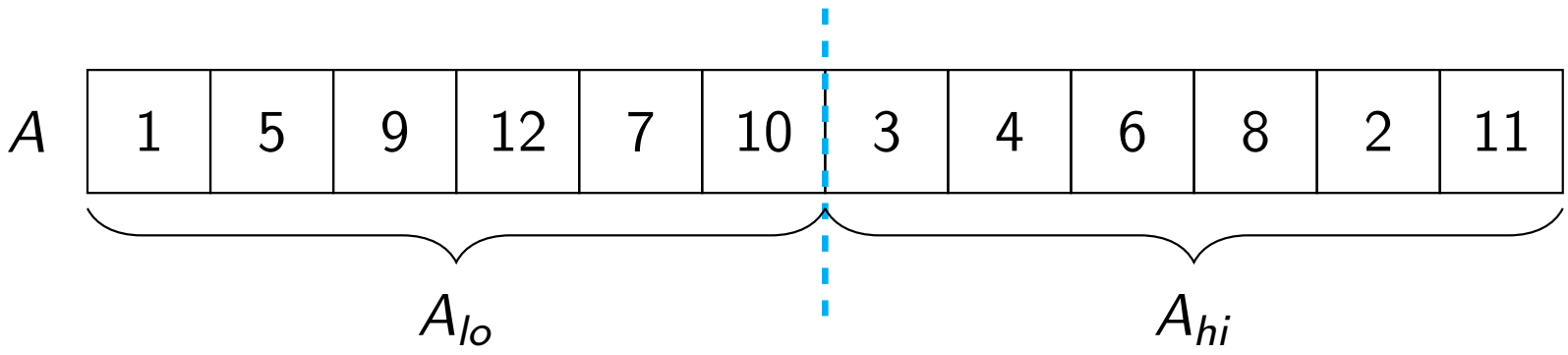
$$1 < 2 \text{ and } a(1) < a(2).$$

However, B prefers movie 4 to 7 and A disagrees, so this pair is an inversion:

$$4 < 7 \text{ and } a(4) < a(7).$$

- A brute force algorithm to count the inversions is to test each pair $i < j$, and add one to the total if $a(i) > a(j)$. Unfortunately this produces a quadratic time algorithm, $T(n) = \Theta(n^2)$.
- We now show that this can be done more efficiently, in time $O(n \log n)$, by applying a divide-and-conquer strategy.
- Clearly, since the total number of pairs is quadratic in n, we cannot afford to count the inversions one-by-one.
- The main idea is to tweak the merge sort algorithm, by extending it to recursively both sort an array A **and** determine the number of inversions in A.

- We split the array A into two (approximately) equal parts $A_{lo} = A[1..m]$ and $A_{hi} = A[m+1..n]$, where $m = \lfloor n/2 \rfloor$.
- Note that the total number of inversions in array A is the sum of:
  - the number of inversions $I(A_{lo})$ in A_{lo} (such as 9 and 7),
  - the number of inversions $I(A_{hi})$ in A_{hi} (such as 4 and 2), and
  - the number of inversions $I(A_{lo}, A_{hi})$ across the two halves (such as 7 and 4).



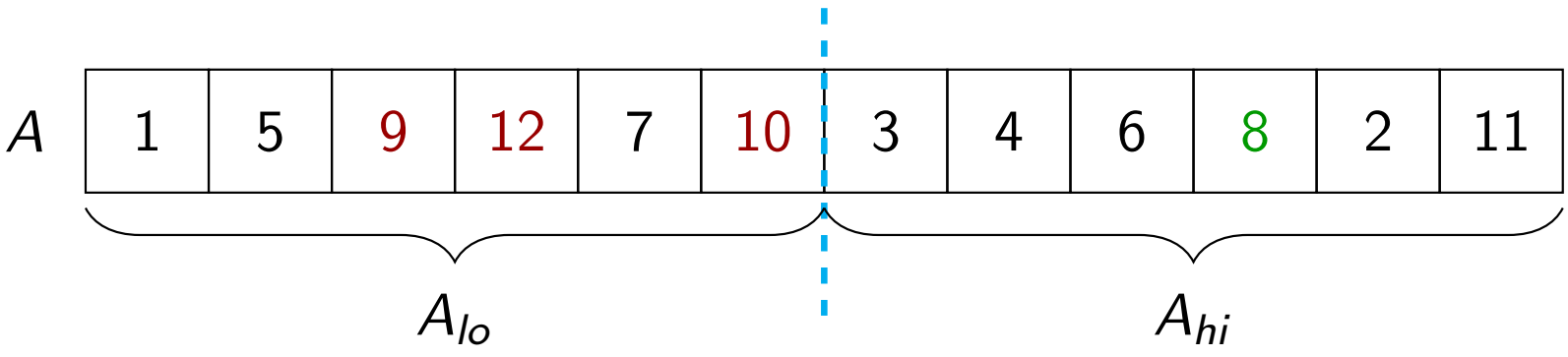
- We have

$$I(A) = I(A_{lo}) + I(A_{hi}) + I(A_{lo}, A_{hi}).$$

- The first two terms of the right-hand side are the number of inversions within A_{lo} and within A_{hi}, which can be calculated recursively.

- The main challenge is to evaluate the last term, which requires us to count the inversions which cross the partition between the two sub-arrays.

- In our example, how many of these cross-partition inversions involve the 8?

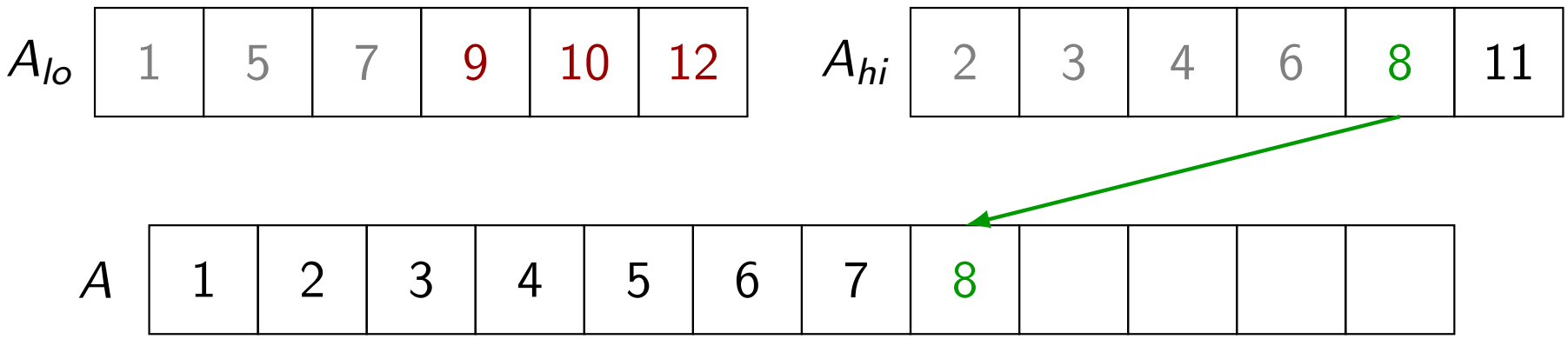


- It's the number of elements of A_{lo} which are greater than 8.
- How would one compute this systematically?

- The idea is to not only count inversions across the partition, but also sort the array. We can then assume that the subarrays A_{lo} and A_{hi} are sorted in the process of counting $I(A_{lo})$ and $I(A_{hi})$.

- We proceed to count $I(A_{lo}, A_{hi})$ (specifically, counting each inversion according to the lesser of its elements) and simultaneously merge as in merge sort.

Each time we reach an element of A_{hi}, we have inversions between this number and each of the remaining elements in A_{lo}. We therefore add the number of elements remaining in A_{lo} to the answer.



$$\text{count} = 5 + 5 + 5 + 4 + 3 + 1 = 23.$$

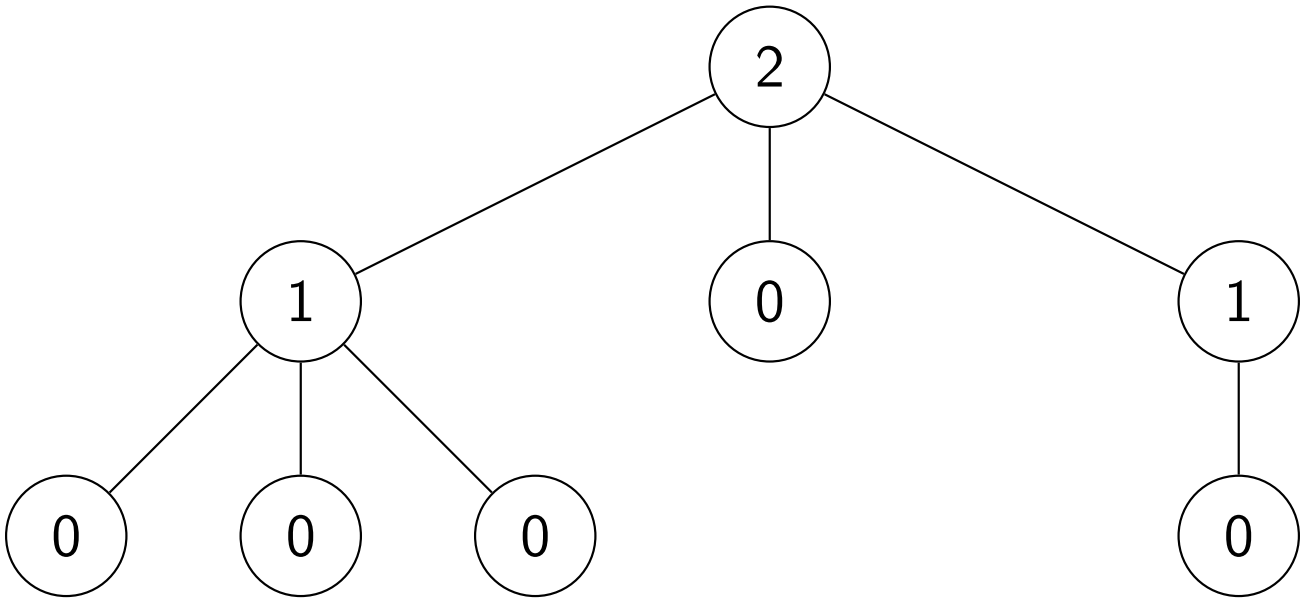
- On the other hand, when we reach an element of A_{lo}, all inversions involving this number have already been counted.
- We have therefore counted the number of inversions within each subarray ($I(A_{lo})$ and $I(A_{hi})$) recursively as well as the number of inversions across the partition ($I(A_{lo}, A_{hi})$), and adding these gives $I(A)$ as required.
- Our modified merge still takes linear time, so this algorithm has the same complexity as merge sort, i.e. $\Theta(n \log n)$.
- **Next:** we look to generalise the divide and conquer method.

Table of Contents

- Steps:
  - **Divide:** Choose a pivot and partition the array around it ($\Theta(n)$)
  - **Conquer:** Sort both sides of the pivot recursively
  - **Combine:** Pass the answer up the recursion tree ($\Theta(1)$)
- *Typically,* recursion is $\log_2 n$ levels deep, with a total of $\Theta(n)$ time spent in each level.
- Time complexity is $\Theta(n \log n)$ in the average case.
- Worst case is $\Theta(n^2)$, with extreme choices of pivot.

Table of Contents

- You may have observed that divide-and-conquer algorithms typically give rise to a tree of subproblems.
  - The root node is the original problem.
  - Splitting a subproblem into several smaller instances corresponds to walking down from a parent node to its children.
  - At a base case, the subproblem cannot be split further, and this is represented by a leaf node.
  - Combining the answers from subproblems corresponds to walking back up to the parent.
- Unsurprisingly perhaps, this is also a natural structure to solve some problems on tree graphs!
  - The subproblem is usually just "what's the answer within the subtree rooted here?"


2
1
0
1
0
0
0
0

- Steps:
  - **Divide:** Iterate over the children of the current node ($\Theta(k)$)
  - **Conquer:** Calculate the heights of the subtrees rooted at each child recursively
  - **Combine:** Take the largest of these values and add one ($\Theta(k)$)
- k here is the number of children of a particular node.
- We have to execute this procedure at n nodes, each taking $\Theta(k)$ time. Now, k is less than n, so the time complexity is $O(n^2)$.
- We can do better!
  - A particular node might have almost n children, but then the others must all have few children.
  - The total number of children across all nodes is $n - 1$.
  - Therefore the time complexity is in fact $\Theta(n)$.

Table of Contents

- **Divide:** split up a large instance into smaller instances of the same type
- **Conquer:** apply the same algorithm to recursively solve each small subproblem, *indepedently of each other*
- **Combine:** synthesise the solution to the original instance from the subproblem solutions
- Base cases are the smallest instances, where no further recursion is possible; usually straightforward.

- Induction on subproblem size.
- If the base cases are solved correctly, **and** the combine step is correct, then subproblems of all sizes are solved correctly, by induction.

Example

In merge sort, the base cases are singleton subarrays, which are already sorted.
Assume that arrays of size up to k are correctly sorted by merge sort. Then a subarray of size $k + 1$ is correctly sorted because:

- its two constituent subarrays are correctly solved, by the earlier assumption, and
- the merge procedure correctly them into the desired sorted array (include proof here if necessary).

Table of Contents

- Recurrences are important to us because they arise in estimations of time complexity of divide-and-conquer algorithms.
- Recall that counting inversions in an array A of size n required us to:
  - recurse on each half of the array (A_{lo} and A_{hi}), and
  - count inversions across the partition, in linear time.
- Therefore the runtime $T(n)$ satisfies

$$T(n) = 2T\left(\frac{n}{2}\right) + c\,n.$$

- Let $a \geq 1$ be an integer and $b > 1$ a real number, and suppose that a divide-and-conquer algorithm:
  - reduces a problem of size n to a many problems of smaller size n/b,
  - with overhead cost of $f(n)$ to split up the problem and combine the solutions from these smaller problems.
- The time complexity of such an algorithm satisfies

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n).$$

Note

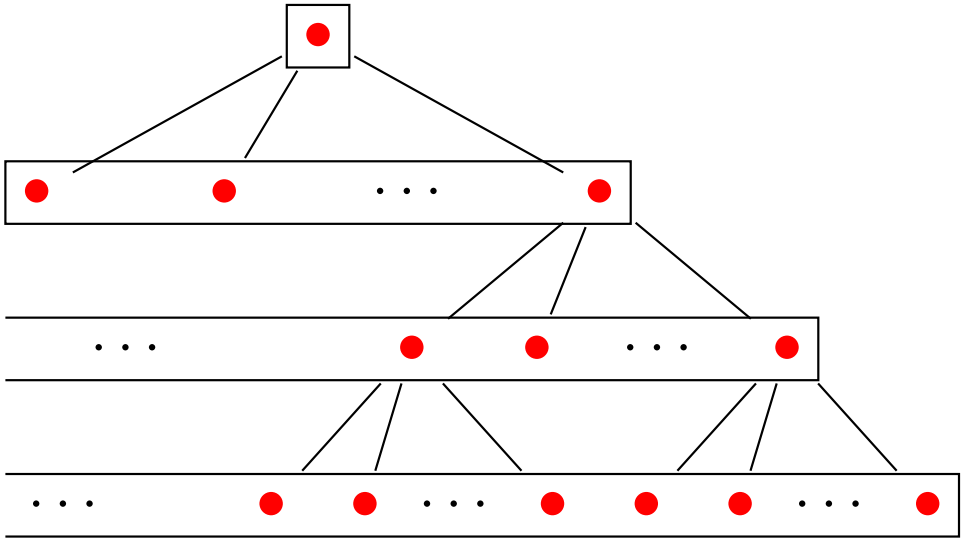
Technically, we should be writing

$$T(n) = a\,T\left(\left\lceil \frac{n}{b} \right\rceil\right) + f(n)$$

but it can be shown that the same asymptotics are achieved if we ignore the rounding and additive constants.

- Some recurrences can be solved explicitly, but this tends to be tricky.
- Fortunately, we *do not* need the exact solution of a recurrence to discuss the efficiency of an algorithm.
- We only need to find:
  - the *growth rate* of the solution i.e., its asymptotic behaviour, and
  - where relevant, the (approximate) *sizes of the constants* involved.
- This is what the **Master Theorem** provides, when it is applicable.
- First, we will build some intuition.

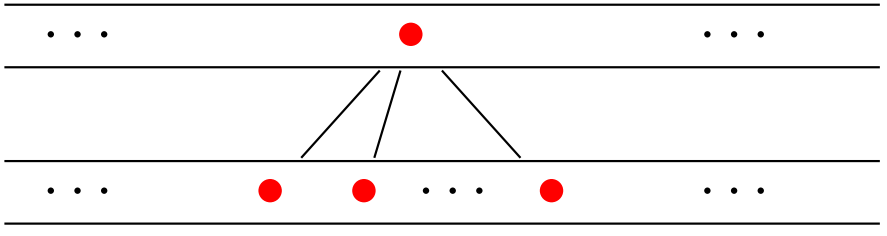
Recurrences of the form $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$



1 instance of size n

a instances of size $\frac{n}{b}$

a^2 instances of size $\frac{n}{b^2}$



a^{k-1} instances of size $\frac{n}{b^{k-1}}$

a^k instances of size $\frac{n}{b^k}$

- What is the depth of the tree?
  - Instances in level k have size n/b^k.
  - This becomes 1 when $k = \log_b n$.
- How many leaves does the tree have?
  - There are a^k instances at depth k, so the number of leaves is $a^{\log_b n}$, which we can rewrite as $n^{\log_b a}$.[2]
- How long is spent at the leaves?
  - Leaf nodes represent base cases, so each takes constant time for a total of $\Theta\left(n^{\log_b a}\right)$.

[2] To see why, try taking $\log_b$ of both expressions.

- As we go down the tree,
  - the number of subproblems grows exponentially, increasing by a factor of a each level, but
  - the size of those subproblems shrinks exponentially, decreasing by a factor of b each level.
- There are three main cases:
  1. If branching by a factor of a outweighs the time saved by doing smaller instances, then nearly all the work is done in the leaves.
  2. If the effects of branching and smaller instances are about equivalent, then each level requires about the same amount of work.
  3. If branching is outweighed by the reduction in instance size, then nearly all the work is done at the root.

Setup

Let:

- $a \geq 1$ be an integer and and $b > 1$ be a real number;
- $f(n) > 0$ be a non-decreasing function defined on the positive integers;
- $T(n)$ be the solution of the recurrence

$$T(n) = a\,T(n/b) + f(n).$$

Define the *critical exponent* $c^* = \log_b a$ and the *critical polynomial* n^{c^*}.[3]

[3] Reminder: $n^{\log_b a}$ is just the number of leaves in the tree.

Theorem

1. If $f(n) = O(n^{c^* - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{c^*})$.

2. If $f(n) = \Theta(n^{c^*} \log^k n)$ for some $k \geq 0$,[4] then $T(n) = \Theta(n^{c^*} \log^{k+1} n)$.

3. If $f(n) = \Omega(n^{c^* + \varepsilon})$ for some $\varepsilon > 0$, **and** for some $k < 1$ and some n_0,

$$a\, f\,(n/b) \leq k\, f(n) \tag{1}$$

holds for all $n > n_0$ (the *regularity condition*), then $T(n) = \Theta(f(n))$.

[4] There are extensions of this case, but we won't use them.

Theorem (continued)

4. If none of these conditions hold, the Master Theorem is not applicable.

Exercise

Prove that $f(n) = \Omega(n^{c^*+\varepsilon})$ is a consequence of (1).

Remark

- Recall that for $a, b > 1$, $\log_a n = \Theta(\log_b n)$, so we can omit the base and simply write statements of the form $f(n) = \Theta(g(n) \log n)$.
- However, $n^{\log_a x}$ is not interchangeable with $n^{\log_b x}$ - the base must be specified in such expressions.

Example 1

Let $T(n) = 4\,T(n/2) + n$.

Then the critical exponent is $c^* = \log_b a = \log_2 4 = 2$, so the critical polynomial is n^2.

Now, $f(n) = n = O(n^{2-\varepsilon})$ for small ε (e.g. 0.1).

This satisfies the condition for case 1, so $T(n) = \Theta(n^2)$.

Example 2

Let $T(n) = 2\,T(n/2) + 5\,n$.

Then the critical exponent is $c^* = \log_b a = \log_2 2 = 1$, so the critical polynomial is n.

Now, $f(n) = 5\,n = \Theta(n)$.

This satisfies the condition for case 2 with $k = 0$, so $T(n) = \Theta(n \log n)$.

Example 3

Let $T(n) = 3\,T(n/4) + n$.

Then the critical exponent is $c^* = \log_4 3 \approx 0.7925$, so the critical polynomial is $n^{\log_4 3}$.

Now, $f(n) = n = \Omega(n^{\log_4 3 + \varepsilon})$ for small ε (e.g. 0.1).

Also, $a\,f\left(\frac{n}{b}\right) = 3f\left(\frac{n}{4}\right) = \frac{3n}{4}\ n\ < k\,n = k\,f(n)$ for $k = 0.9 < 1$.

This satisfies the condition for case 3, so $T(n) = \Theta(f(n)) = \Theta(n)$.

Example 4

Let $T(n) = 2\,T(n/2) + n \log_2(\log_2 n)$.

Then the critical exponent is $c^* = \log_2 2 = 1$, so the critical polynomial is n.

Now, $f(n) = n \log_2(\log_2 n) \neq O(n)$, so the conditions for case 1 and 2 do not apply.

However,

$$f(n) \neq \Omega(n^{1+\varepsilon}),$$

no matter how small we choose $\varepsilon > 0$.

Therefore the Master Theorem does **not** apply!

- In this section, we investigate how to multiply large integers quickly.
- This isn't a very interesting question in the uniform model of computation, where all arithmetic operations (including multiplication) are assumed to take constant time.
- It is much more interesting in the *logarithmic* model of computation, where potentially huge inputs are thought of not as constant sized, but as having size equal to their *width*, i.e. the number of symbols required to express them.
- In base 2, this is the number of bits in the number. An integer of value N has $n = \log_2 N$ bits.
- Recall however that any other base is equivalent, since the difference between logarithms of different bases is a constant factor.

```
 1 0 0 0 1      carry
   1 0 1 0 1     first integer
+  1 1 0 0 1     second integer
  -----------
 1 0 1 1 1 0     result
```

- Adding 3 bits can be done in constant time.
- It follows that the whole algorithm runs in linear time i.e., $O(n)$ many steps.

Question

Can we add two n-bit numbers in faster than in linear time?

Answer

No! There is no asymptotically faster algorithm because we have to use every bit of the input numbers, which takes $\Omega(n)$ time.

```
        1 1 0 1  <- first input integer
      * 1 0 1 0  <- second input integer
        -------
        1 0 1 0  \
      0 0 0 0     \ O(n^2) intermediate operations:
    1 0 1 0       / O(n^2) elementary  multiplications
  1 0 1 0        /    + O(n^2) elementary additions
---------------
1 0 0 0 0 0 1 0  <-  result of length 2n
```

- Multiplying two bits is the same operation as logical AND, so clearly it takes constant time, but the same complexity applies for digits of other bases.
- Thus the above procedure runs in time $O(n^2)$.

Question

Can we multiply two n-bit numbers in linear time, like addition?

Answer

No one knows! "Simple" problems can actually turn out to be difficult!

Question

Can we do it in faster than quadratic time? Let's try divide and conquer.

- Split the two input numbers A and B into halves:
  - A_0, B_0 - the least significant $n/2$ bits;
  - A_1, B_1 - the most significant $n/2$ bits.

$$A = A_1 2^{\frac{n}{2}} + A_0$$
$$B = B_1 2^{\frac{n}{2}} + B_0$$

$$\underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

- AB can now be calculated recursively using the following equation:

$$AB = A_1 B_1 2^n + (A_1 B_0 + B_1 A_0) 2^{\frac{n}{2}} + A_0 B_0.$$

Algorithm

We recursively apply the following algorithm to multiply two integers A and B, each consisting of n bits.

The base case is $n = 1$, where we simply evaluate the product.

Otherwise, we let

- A_1 and A_0 be the most and least significant parts of A respectively
- B_1 and B_0 be the most and least significant parts of A respectively.

Algorithm (continued)

Now, we compute

- $X = A_0 B_0$
- $Y = A_0 B_1$
- $Z = A_1 B_0$
- $W = A_1 B_1$

recursively, as each is a product of two $n/2$-bit integers.

Finally, we compute $W\,2^n + (Y + Z)\,2^{n/2} + X$ by summing

- W shifted left by n bits
- Y and Z each shifted left by $n/2$ bits, and
- X with no shift.

- How many steps does this algorithm take?
- Each multiplication of two n digit numbers is replaced by four multiplications of $n/2$ digit numbers

$$X = A_0 B_0, Y = A_0 B_1, Z = A_1 B_0, W = A_1 B_1,$$

  plus we have a **linear** overhead to shift and add.

- Therefore the recurrence is

$$T(n) = 4\,T\left(\frac{n}{2}\right) + c\,n.$$

- Let's use the Master Theorem!

- From the recurrence

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

  we have $a = 4$, $b = 2$ and $f(n) = c\,n$.

- The critical exponent is $c^* = \log_2 4 = 2$, so the critical polynomial is n^2.

- Then $f(n) = c\,n = O(n^{2-0.1})$, so Case 1 applies.

- We conclude that $T(n) = \Theta(n^{c^*}) = \Theta(n^2)$, i.e., we gained **nothing** with our divide-and-conquer!

Question

Is there a smarter multiplication algorithm taking less than $O(n^2)$ many steps?

Answer

Remarkably, there is!

Table of Contents

- In 1952, one of the most famous mathematicians of the 20^{th} century, Andrey Kolmogorov, conjectured that you cannot multiply in less than quadratic many elementary operations.

- In 1960, Anatoly Karatsuba, then a 23-year-old student, found an algorithm (later it was called "divide-and-conquer") that multiplies two n-digit numbers in $\Theta\left(n^{\log_2 3}\right) \approx \Theta(n^{1.58\ldots})$ elementary steps, thus disproving the conjecture!! Kolmogorov was shocked!

- Once again we split each of our two input numbers A and B into halves:

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad \underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}.$$
$$B = B_1 2^{\frac{n}{2}} + B_0$$

- Previously we saw that

$$AB = A_1 B_1 2^n + (A_1 B_0 + A_0 B_1) 2^{\frac{n}{2}} + A_0 B_0,$$

but rearranging the bracketed expression gives

$$AB = A_1 B_1 2^n + ((A_1 + A_0)(B_1 + B_0) - A_1 B_1 - A_0 B_0) 2^{\frac{n}{2}} + A_0 B_0,$$

saving one multiplication at each round of the recursion!

Algorithm

We recursively apply the following algorithm to multiply two integers A and B, each consisting of n bits.

The base case is $n = 1$, where we simply evaluate the product.

Otherwise, we let

- A_1 and A_0 be the most and least significant parts of A respectively
- B_1 and B_0 be the most and least significant parts of A respectively.

Algorithm (continued)

Now, we compute

- $X = A_0 B_0$
- $W = A_1 B_1$
- $V = (A_1 + A_0)(B_1 + B_0)$

recursively, as each is a product of two (approximately) $n/2$-bit integers.

Finally, we compute $W\,2^n + (V - W - X)\,2^{n/2} + X$ by summing

- W shifted left by n bits
- V less W and X shifted left by $n/2$ bits, and
- X with no shift.

- How many steps does this algorithm take?
- Each multiplication of two n digit numbers is replaced by three multiplications of $n/2$ digit numbers

$$X = A_0 B_0, W = A_1 B_1, V = (A_1 + A_0)(B_1 + B_0)$$

with linear additional work to shift and add.

- Therefore the recurrence is

$$T(n) = 3T\left(\frac{n}{2}\right) + c\,n.$$

- From the recurrence

$$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n,$$

we have $a = 3$, $b = 2$ and $f(n) = c\,n$.

- Now the critical exponent is $c^* = \log_2 3$. Once again, we are in Case 1 of the Master Theorem, but this time

$$\begin{aligned} T(n) &= \Theta\left(n^{\log_2 3}\right) \\ &= \Theta\left(n^{1.58\ldots}\right) \\ &\neq \Omega(n^2), \end{aligned}$$

disproving Kolmogorov's conjecture.

- We can do even better, by splitting the input numbers A and B into more than two pieces.
- In fact, for any $\epsilon > 0$, we can achieve $T(n) = O\left(n^{1+\epsilon}\right)$.
- However, with numbers divided into $p+1$ pieces, the constant factors involved are on the order of p^p, so it is only worthwhile for extremely large inputs.
- Moral: in practice, we can only rely on asymptotic estimates if the constants hidden by the big-oh notation are known to be reasonably small!

Breakthrough

Harvey and van der Hoeven (2021) gave an $O(n \log n)$ algorithm,[5] although it may only be faster than existing methods for astronomically large numbers.

[5] You can find the paper here, and media coverage and FAQ here.

- If we split the inputs into p pieces, the problem is then to multiply

$$A = A_{p-1}2^{(p-1)k} + \ldots + A_2 2^{2k} + A_1 2^k + A_0$$

and

$$B = B_{p-1}2^{(p-1)k} + \ldots + B_2 2^{2k} + B_1 2^k + B_0,$$

where $k = \frac{n}{p}$ is the width of each piece.

- The key observation is that $2^{2k}, \ldots, 2^{(p-1)k}$ are the powers of 2^k, but there is otherwise nothing special about powers of 2.

- We could instead work on the more general problem of multiplying *polynomials*

$$P_A(x) = A_{p-1}x^{p-1} + \ldots + A_2x^2 + A_1x + A_0$$

and

$$P_B(x) = B_{p-1}x^{p-1} + \ldots + B_2x^2 + B_1x + B_0.$$

- The objective is to compute the coefficients of $P_A(x)P_B(x)$ using as few multiplications as possible.
- It turns out this is a very important (perhaps *the most* important) practical problem!
- Not limited to integer coefficients.

Table of Contents

Definition

A polynomial is a function of the form

$$P(x) = A_n x^n + A_{n-1} x^{n-1} + \ldots + A_1 x + A_0.$$

The a_i are called *coefficients*, and n is the *degree*.
The coefficients can be any kind of number, not just integers.

Note

In this section, we will be working in a model of computation where the coefficients and input values can be added and multiplied in constant time. We'll see that this often involves real and even complex number arithmetic, so in practice it would be approximated using floating-point numbers.
Accordingly, the width of the coefficients isn't a relevant quantity anymore, so we will reintroduce variable n with a different meaning.

- **Addition:** given two polynomials P_A and P_B, construct the sum $P_A + P_B$ (which is itself a polynomial).
- **Evaluation:** given a polynomial P and an input x_0, find the value of $P(x_0)$.
- **Multiplication:** given two polynomials P_A and P_B, construct the product $P_A P_B$ (which is itself a polynomial).

- The coefficient representation

$$P(x) = A_n x^n + A_{n-1} x^{n-1} + \ldots + A_1 x + A_0$$

  allows addition term-by-term in $O(n)$.
- Evaluation appears complicated because of the powers. Horner's rule!
  - Other than A_0, all terms have at least one factor of x, so

$$P(x) = A_0 + x(A_1 + A_2 x + \ldots + A_{n-1} x^{n-2} + A_n x^{n-1}).$$

  - Now the same situation applies within the brackets! Continuing thus,

$$P(x) = A_0 + x(A_1 + x(A_2 + x(\ldots))).$$

  - Evaluate from the middle: start with A_n, multiply by x, add A_{n-1}, multiply by x, and so on. $O(n)$ time.

- The main weakness of the coefficient representation is multiplication, which takes quadratic time.

- Distributing the product

$$(A_n x^n + \ldots + A_1 x + A_0)(B_n x^n + \ldots + B_1 x + B_0)$$

  involves multiplying every pair $A_i B_j$.

- Is there an alternative representation that could be more useful for multiplication?

Value representation

- We can instead represent a polynomial of degree n by a sequence of $n+1$ values!
  - There is a unique line through any two points in the plane, a unique parabola through any three points, and so on.
- Fix a selection of inputs $x_0, \ldots, x_n$. Then the corresponding values $P(x_0), \ldots, P(x_n)$ represent a unique polynomial of degree up to n, just as much as the coefficient representation would.
- There isn't a straightforward way to add two polynomials in the value representation, nor to evaluate at an input which isn't among the x_i.
- But multiplication now takes linear time; just multiply pointwise: $(P_A P_B)(x_0) = P_A(x_0) P_B(x_0)$.
- Note: we'll need enough points to uniquely determine the product polynomial $(A_n x^n + \ldots)(B_n x^n + \ldots)$.

1. Evaluate (quickly, somehow) $P_A(x)$ and $P_B(x)$ at $2n+1$ distinct points

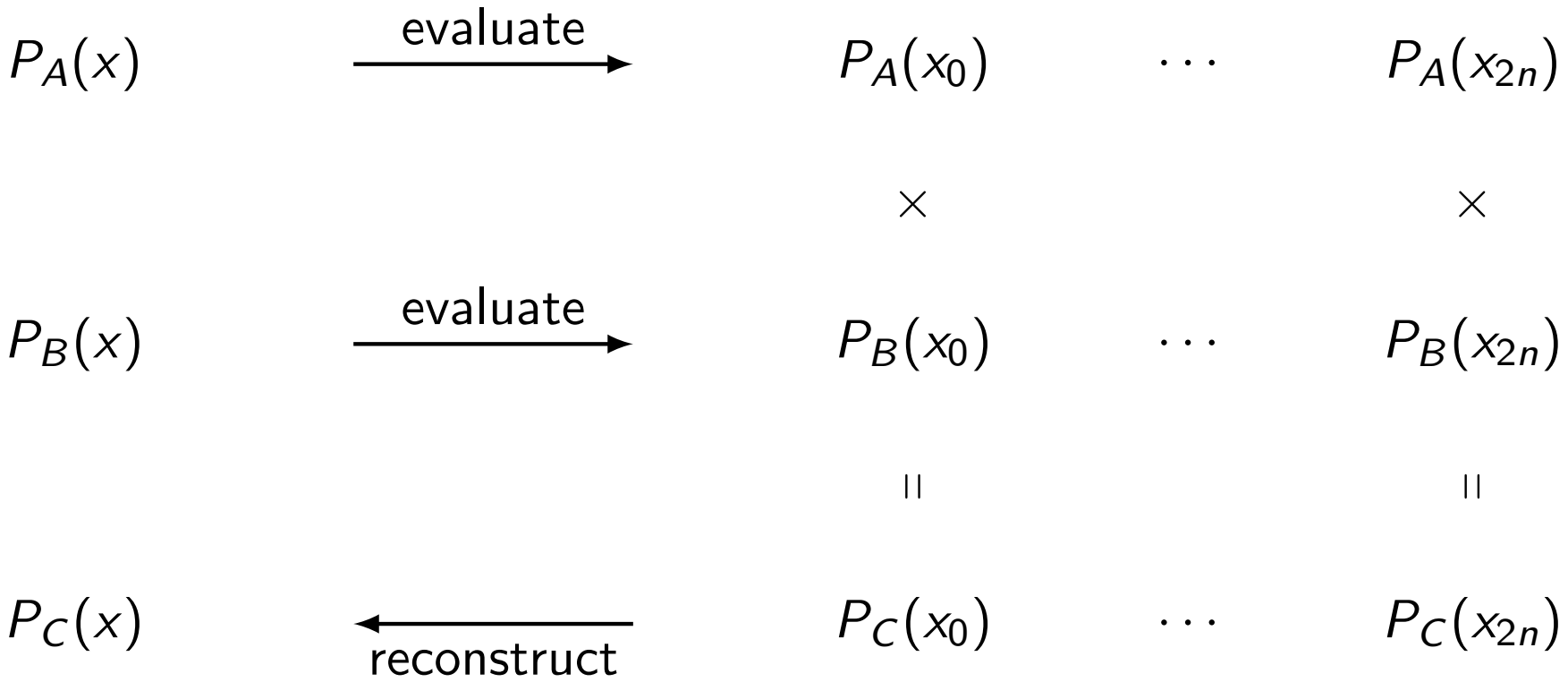
$$x_0, x_1, \ldots, x_{2n}$$

2. Multiply them point by point using $2n+1$ multiplications of large numbers, to get $2n+1$ values of the product polynomial $P_C(x)$

3. Reconstruct (quickly, somehow) the coefficients of $P_C(x)$, i.e.

$$P_C(x) = C_{2n}x^{2n} + C_{2n-1}x^{2n-1} + \ldots + C_1x + C_0.$$

Our strategy to multiply polynomials fast



- What inputs $x_0, \ldots, x_{2n}$ should we choose?
- $-n, \ldots, 0, \ldots, n$ seems like a natural choice, but then the polynomials take values on the order of n^n.

Question

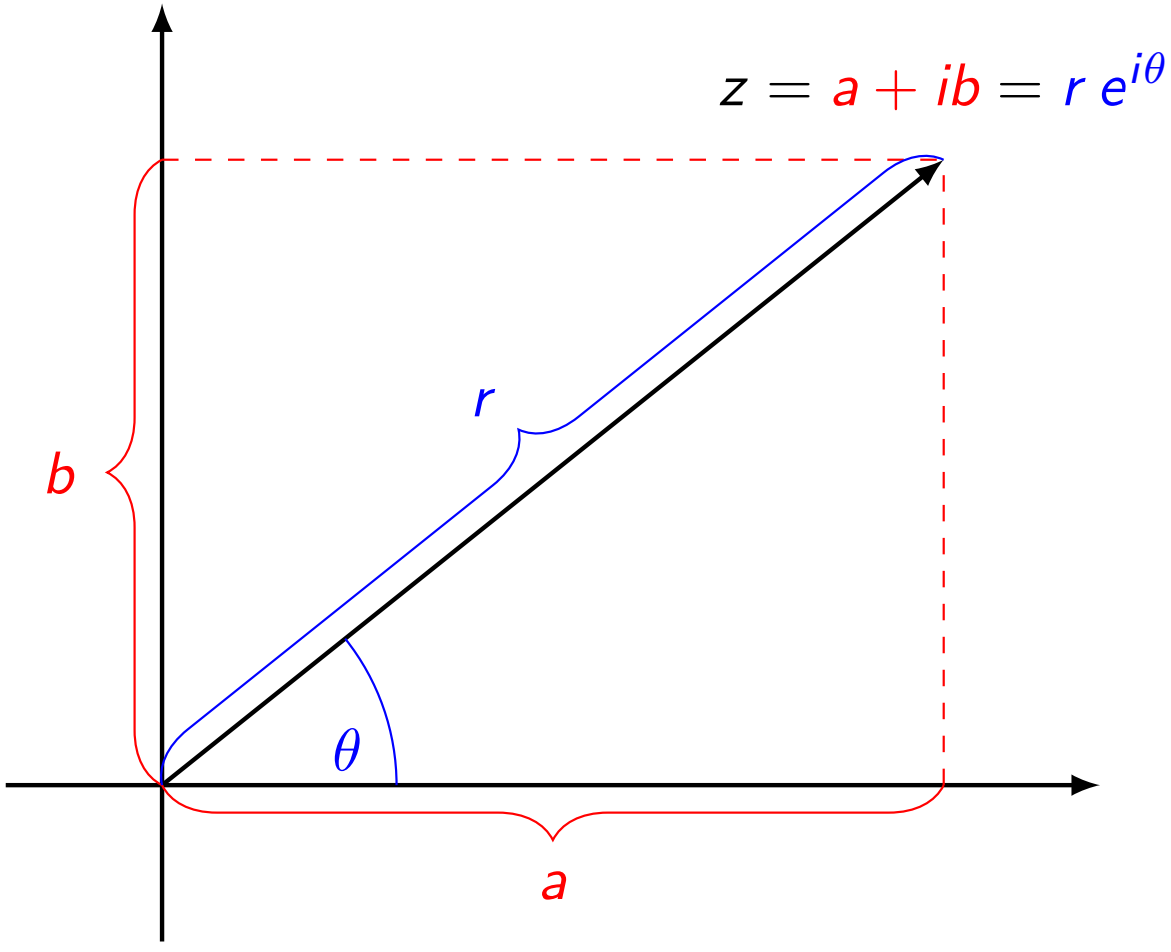
What values should we take for $x_0, \ldots, x_{2n}$ to avoid "explosion" of size when we evaluate x_i^n while computing $P_A(x_i) = A_0 + A_1 x + \ldots + A_n x_i^n$?

Answer

We would like $|x_i^n| = 1$, but this can't be achieved with only real numbers ...

- Let $z = a + ib$ be a complex number, where $a, b \in \mathbb{R}$.
- Then we can define the
  - *modulus* $|z|$, a positive real number r such that $r = \sqrt{a^2 + b^2}$ and
  - *argument* $\arg z$, an angle $\theta \in (-\pi, \pi]$ such that $a = r\cos\theta$ and $b = r\sin\theta$.
- Then we have

$$\begin{aligned} z &= a + ib \\ &= r(\cos\theta + i\sin\theta) \\ &= r\,e^{i\theta}. \end{aligned}$$


$z = a + ib = r\,e^{i\theta}$
r
b
θ
a

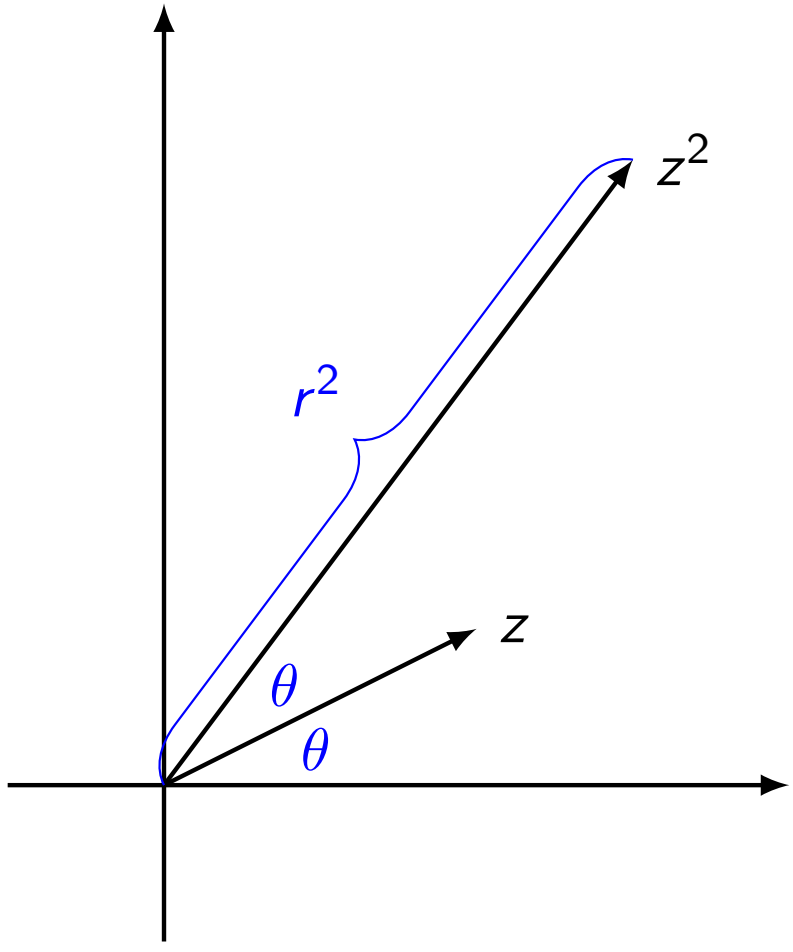
To multiply two complex numbers, we multiply the moduli and add the arguments.

Theorem

If $z = r\,e^{i\theta}$ and $w = s\,e^{i\phi}$, then $z\,w = (r\,s)\,e^{i\,(\theta+\phi)}$.

Corollary

In particular, if $z = r\,e^{i\theta}$ then $z^n = r^n\,e^{i\,(n\theta)}$.


z^2
r^2
z
θ
θ

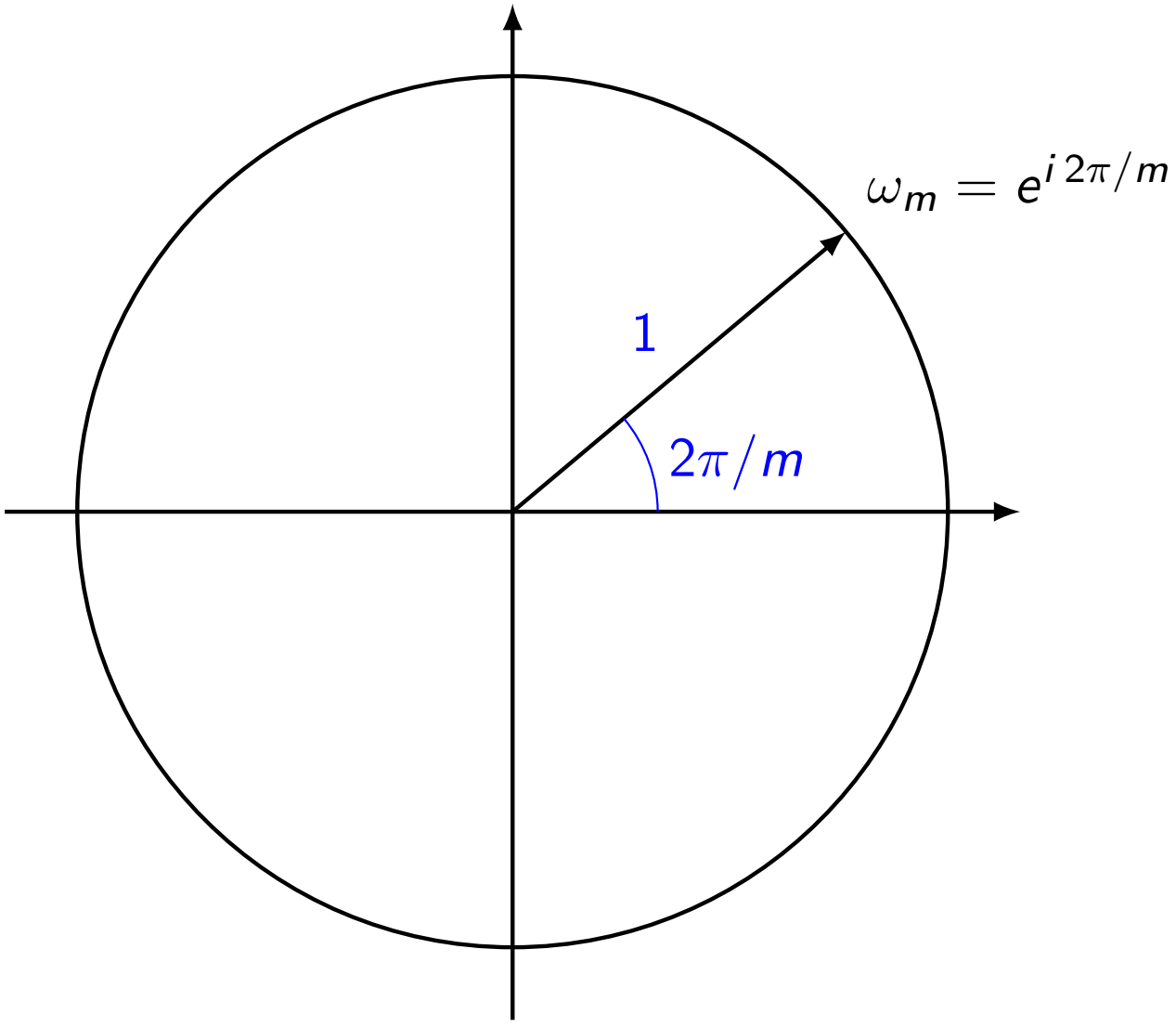
- We are interested in the solutions of $z^m = 1$, known as *roots of unity of order m*.
- Solving $r^m\, e^{i\,(m\theta)} = 1$, we see that

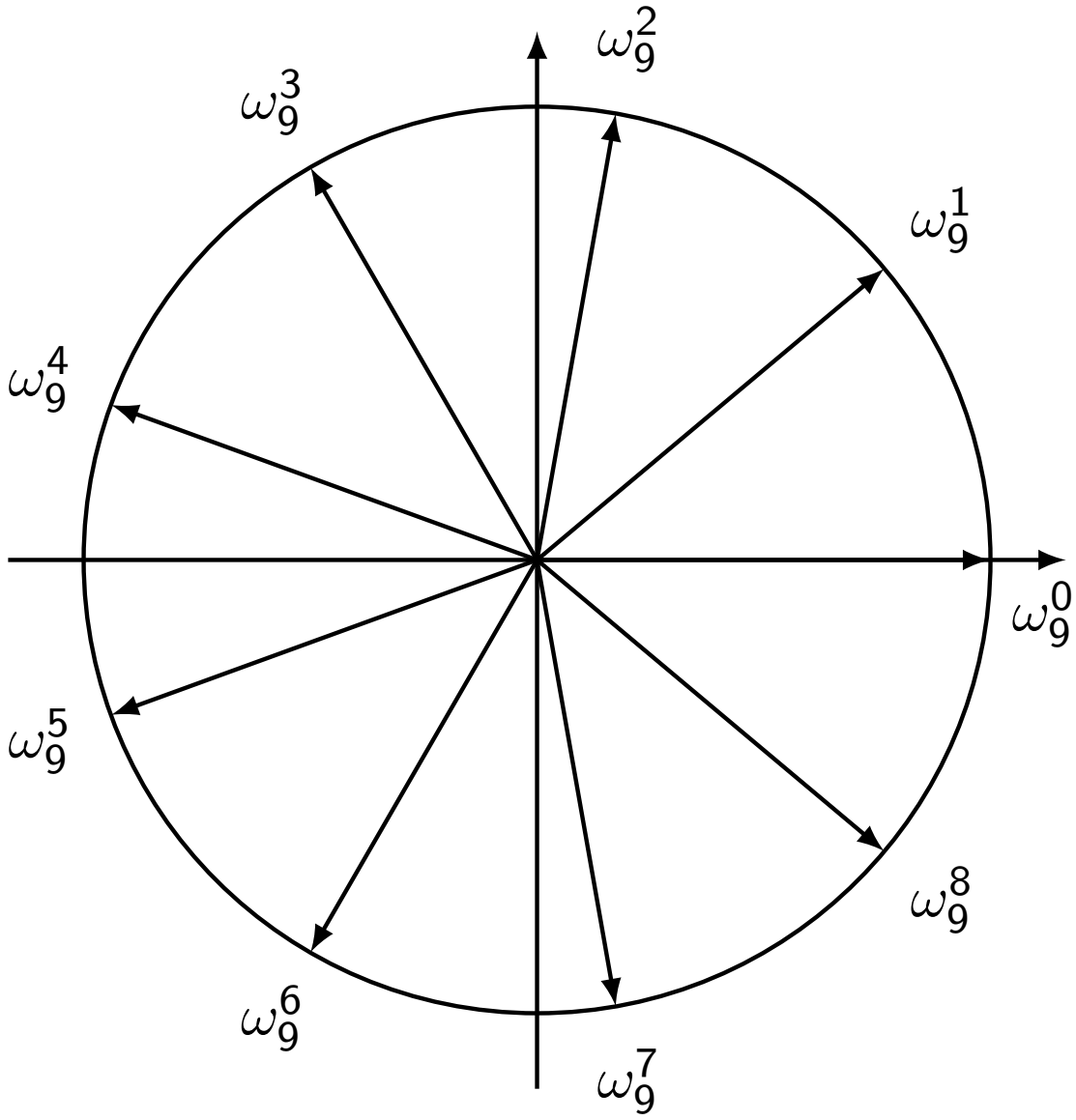
$$r^m = 1, \text{ i.e. } r = 1$$

and

$$m\theta = 2\pi\, k \text{ i.e. } \theta = \frac{2\pi\, k}{m}.$$

- Let $\omega_m = e^{i\,2\pi/m}$. Then $z = \omega_m^k$, i.e., all roots of unity of order m can be written as powers of ω_m. We say that ω_m is a *primitive root of unity of order m*.
- Note that there are only m distinct values here, as $\omega_m^m = \omega_m^0$.


$\omega_m = e^{i\,2\pi/m}$
1
$2\pi/m$


ω_9^2
ω_9^3
ω_9^1
ω_9^4
ω_9^0
ω_9^5
ω_9^8
ω_9^6
ω_9^7

- For $\omega_m = e^{i\,2\pi/m}$ and for all k such that $0 \leq k \leq m-1$,

$$((\omega_m)^k)^m = (\omega_m)^{m\,k} = ((\omega_m)^m)^k = 1^k = 1.$$

- Thus, $\omega_m^k = (\omega_m)^k$ is also a root of unity.

Theorem

ω_m^k is a primitive root of unity of order m if and only if $\gcd(m, k) = 1$.

- The product of two roots of unity of order m is given by

$$\omega_m^i \omega_m^j = \omega_m^{i+j},$$

  which is itself a root of unity of the same order.

- So the set of all roots of unity of order m, i.e., $\{1, \omega_m, \omega_m^2, \ldots, \omega_m^{m-1}\}$ is closed under multiplication (and by extension, under taking powers).

- Note that this is not true for addition, i.e., the sum of two roots of unity is NOT another root of unity!

Note

This is an example of a *group*.

Cancellation Lemma

For all positive integers ℓ, m and integers k, $\omega_{\ell m}^{\ell k} = \omega_m^k$.

Proof

$$\omega_{\ell m}^{\ell k} = \left(e^{i\frac{2\pi}{\ell m}}\right)^{\ell k} = e^{i\frac{2\pi \ell k}{\ell m}} = e^{i\frac{2\pi k}{m}} = \left(e^{i\frac{2\pi}{m}}\right)^k = \omega_m^k.$$

- In particular, $(\omega_{2m}^k)^2 = \omega_{2m}^{2k} = (\omega_{2m}^2)^k = \omega_m^k$, i.e. the squares of the roots of unity of order $2m$ are just the roots of unity of order m.

- Given two polynomials of degree at most n,

$$P_A(x) = A_n x^n + \ldots + A_0 \text{ and } P_B(x) = B_n x^n + \ldots + B_0,$$

we want to find $P_C(x) = P_A(x)\,P_B(x)$.

1. Evaluate $P_A(x)$ and $P_B(x)$ at $m = 2n+1$ distinct points

$$\omega_m^0, \omega_m^1, \ldots, \omega_m^{m-1}$$

2. Multiply them point by point using $2n+1$ multiplications of large numbers, to get $2n+1$ values of $P_C(x)$
3. Reconstruct the coefficients of $P_C(x)$, i.e.

$$P_C(x) = C_{2n} x^{2n} + C_{2n-1} x^{2n-1} + \ldots + C_1 x + C_0.$$

$$
\begin{array}{ccccc}
P_A(x) & \xrightarrow{\text{evaluate}} & P_A\left(\omega_m^0\right) & \cdots & P_A\left(\omega_m^{m-1}\right) \\
 & & \times & & \times \\
P_B(x) & \xrightarrow{\text{evaluate}} & P_B\left(\omega_m^0\right) & \cdots & P_B\left(\omega_m^{m-1}\right) \\
 & & = & & = \\
P_C(x) & \xleftarrow[\text{reconstruct}]{} & P_C\left(\omega_m^0\right) & \cdots & P_C\left(\omega_m^{m-1}\right)
\end{array}
$$

Definition

Let $A = \langle A_0, A_1, \ldots, A_{m-1} \rangle$ be a sequence of m real or complex numbers, and let the corresponding polynomial be $P_A(x) = \sum_{j=0}^{m-1} A_j x^j$.

Let $\widehat{A}_k = P_A(\omega_m^k)$ for all $0 \leq k \leq m-1$, the values of P_A at the roots of unity of order m.

The sequence of values $\widehat{A} = \langle \widehat{A}_0, \widehat{A}_1, \ldots, \widehat{A}_{m-1} \rangle$ is called the *Discrete Fourier Transform* (DFT) of A.

The DFT is just the value representation at the mth roots of unity.

- Our polynomials $P_A(x)$ and $P_B(x)$ have degree n, so the corresponding sequences of coefficients have only $n+1$ terms.
- However, we defined the DFT of a sequence as having the same length as the original sequence, and we must obtain the values at all m roots of unity of order m.
- This is easily rectified by padding the sequences A and B with zeros at the end, corresponding to the terms $0x^{n+1} + 0x^{n+2} + \ldots + 0x^{2n}$, since $m = 2n+1$.
- The DFT $\widehat{A}$ of a sequence A can be computed *very fast* using a divide-and-conquer algorithm called the Fast Fourier Transform.

- We can now compute the DFTs of the two (0 padded) sequences:

$$\mathrm{DFT}(\langle A_0, A_1, \ldots, A_n, \underbrace{0, \ldots, 0}_{n}\rangle) = \langle \widehat{A}_0, \widehat{A}_1, \ldots, \widehat{A}_{m-1}\rangle$$

  and similarly for B.
- For each k we multiply the corresponding values $\widehat{A}_k = P_A(\omega_m^k)$ and $\widehat{B}_k = P_B(\omega_m^k)$, thus obtaining

$$\widehat{C}_k = \widehat{A}_k \widehat{B}_k = P_A(\omega_m^k) P_B(\omega_m^k) = P_C(\omega_m^k)$$

- We then use the inverse transformation for DFT, called IDFT, to recover the coefficients $\langle C_0, C_1, \ldots, C_{m-1}\rangle$ of the product polynomial $P_C(x)$ from the sequence $\langle \widehat{C}_0, \widehat{C}_1, \ldots, \widehat{C}_{m-1}\rangle$ of its values $C_k = P_C(\omega_m^k)$ at the roots of unity of order m.

Our strategy to multiply polynomials fast

$$\langle A_0, A_1, \ldots, A_n, \underbrace{0, \ldots, 0}_{n} \rangle \xrightarrow{\text{DFT}} \widehat{A}_0 \quad \cdots \quad \widehat{A}_{m-1}$$

$$\times \qquad\qquad \times$$

$$\langle B_0, B_1, \ldots, B_n, \underbrace{0, \ldots, 0}_{n} \rangle \xrightarrow{\text{DFT}} \widehat{B}_0 \quad \cdots \quad \widehat{B}_{m-1}$$

$$\| \qquad\qquad \|$$

$$\langle C_0, C_1, \ldots, C_{m-1} \rangle \xleftarrow{\text{IDFT}} \widehat{C}_0 \quad \cdots \quad \widehat{C}_{m-1}$$

Question

How long does it take to compute the DFT of a sequence A, i.e. find the values

$$P_A(\omega_m^k) = A_0 + A_1\,\omega_m^k + A_2\,(\omega_m^k)^2 + \ldots + A_{m-1}\,(\omega_m^k)^{m-1}$$

for all k such that $0 \leq k < m$?

Answer

Even if we were to precompute all powers of ω_m (recall, there are only m distinct values), there are m multiplications required for each of m values $P_A(\omega_m^k)$.

However, we can use divide-and-conquer to find all the required values in $O(n \log n)$ time!

- We can assume that m is a power of 2 - otherwise we can pad $P_A(x)$ with zero coefficients until its number of coefficients becomes equal to the nearest power of 2.

Exercise

Show that for every m which is not a power of two the smallest power of 2 larger or equal to m is smaller than $2m$.

Hint

Consider m in binary. How many bits does the nearest power of two have?

Problem

Given a sequence $A = \langle A_0, A_1, \ldots, A_{m-1} \rangle$, compute its DFT, i.e. find the values $P_A(x)$ at $x = \omega_m^k$ for all k such that $0 \leq k < m$.

The main idea

We use divide-and-conquer by splitting the polynomial $P_A(x)$ into the even powers and the odd powers:

$$\begin{aligned} P_A(x) &= (A_0 + A_2\,x^2 + A_4\,x^4 + \ldots + A_{m-2}\,x^{m-2}) \\ &\quad + (A_1\,x + A_3\,x^3 + \ldots + A_{m-1}\,x^{m-1}) \\ &= \left(A_0 + A_2\,x^2 + A_4\,(x^2)^2 + \ldots + A_{m-2}\,(x^2)^{m/2-1}\right) \\ &\quad + x\left(A_1 + A_3\,x^2 + A_5\,(x^2)^2 + \ldots + A_{m-1}\,(x^2)^{m/2-1}\right) \end{aligned}$$

- Let us define $A^{[0]} = \langle A_0, A_2, A_4, \ldots A_{m-2} \rangle$ and $A^{[1]} = \langle A_1, A_3, A_5, \ldots A_{m-1} \rangle$, so that

$$P_{A^{[0]}}(y) = A_0 + A_2\, y + A_4\, y^2 + \ldots + A_{m-2}\, y^{m/2-1}$$
$$P_{A^{[1]}}(y) = A_1 + A_3\, y + A_5\, y^2 + \ldots + A_{m-1}\, y^{m/2-1}$$

  and hence

$$P_A(x) = P_{A^{[0]}}(x^2) + x\, P_{A^{[1]}}(x^2).$$

- Note that the polynomials $P_{A^{[0]}}(y)$ and $P_{A^{[1]}}(y)$ have $m/2$ coefficients each, while the polynomial $P_A(x)$ has m coefficients.

Question

Have we successfully reduced a problem of size m to a problem of size $m/2$?

- **Problem of size m:**
  Evaluate a polynomial with m coefficients at all roots of unity of order m.

- **Problem of size $m/2$:**
  Evaluate a polynomial with $m/2$ coefficients at all roots of unity of order $m/2$.

- The original problem was to evaluate the polynomial $P_A(x)$ with m coefficients at inputs $x = \omega_m^k$ for each $0 \leq k < m$. We have reduced it to the evaluation of two polynomials $P_{A^{[0]}}(y)$ and $P_{A^{[1]}}(y)$ each with $m/2$ coefficients at points

$$y = x^2 = \omega_m^{2k}.$$

- Recall that we assumed m is a power of 2, and hence even. We can therefore use the cancellation lemma

$$y = \omega_m^{2k} = \omega_{m/2}^k$$

to deduce that y is a root of unity of order $m/2$.

- Note that k goes up to $m-1$, but there are only $m/2$ distinct roots of unity of order $m/2$. Since $\omega_{m/2}^{m/2} = 1$, we can easily simplify the later terms as repetitions of the earlier terms.

- Therefore we can write

$$P_A(\omega_m^k) = P_{A^{[0]}}\left((\omega_m^k)^2\right) + \omega_m^k \cdot P_{A^{[1]}}\left((\omega_m^k)^2\right)$$
$$= P_{A^{[0]}}\left(\omega_{m/2}^k\right) + \omega_m^k \cdot P_{A^{[1]}}\left(\omega_{m/2}^k\right).$$

- Thus, we have reduced a problem of size m to two such problems of size $m/2$, plus a linear overhead to multiply the ω_m^k terms.

- We have recursively reduced evaluation of a polynomial $P_A(x)$ with m coefficients at m roots of unity of order m to evaluations of two polynomials $P_{A^{[0]}}(y)$ and $P_{A^{[1]}}(y)$, each with $m/2$ coefficients, at $m/2$ many roots of unity of order $m/2$.

- Once we get these $m/2$ values of $P_{A^{[0]}}(y)$ and $P_{A^{[1]}}(y)$ we need m additional multiplications to obtain the values of

$$\underbrace{P_A\left(\omega_m^k\right)}_{y_k} = \underbrace{P_{A^{[0]}}\left(\omega_{m/2}^k\right)}_{y_k^{[0]}} + \omega_m^k \underbrace{P_{A^{[1]}}\left(\omega_{m/2}^k\right)}_{y_k^{[1]}}$$

for all $0 \leq k < m$.

- Thus, we have reduced a problem of size m to two such problems of size $m/2$, plus a linear overhead.

- Consequently, our algorithm's run time satisfies the recurrence

$$T(m) = 2\,T\left(\frac{m}{2}\right) + c\,m.$$

- Case 2 of the Master Theorem gives

$$T(m) = \Theta(m \log m) = \Theta(n \log n).$$

- The evaluation of a polynomial $P_A(x) = A_0 + A_1 x + \ldots + A_{m-1}x^{m-1}$ at roots of unity ω_m^k of order m can be represented using the following matrix-vector product.

$$\underbrace{\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_m & \omega_m^2 & \cdots & \omega_m^{m-1} \\ 1 & \omega_m^2 & \omega_m^{2\cdot 2} & \cdots & \omega_m^{2\cdot(m-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_m^{m-1} & \omega_m^{2(m-1)} & \cdots & \omega_m^{(m-1)(m-1)} \end{pmatrix}}_{M} \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ \vdots \\ A_{m-1} \end{pmatrix} = \begin{pmatrix} P_A(1) \\ P_A(\omega_m) \\ P_A(\omega_m^2) \\ \vdots \\ P_A(\omega_m^{m-1}) \end{pmatrix}$$

- The FFT is just a method of replacing this matrix-vector multiplication taking m^2 many multiplications with a $\Theta(m \log m)$ procedure.

- We also need a way to 'reconstruct' the coefficients from these values, i.e. perform the inverse DFT.

- If M is invertible, we could write:

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ \vdots \\ A_{m-1} \end{pmatrix} = M^{-1} \begin{pmatrix} P_A(1) \\ P_A(\omega_m) \\ P_A(\omega_m^2) \\ \vdots \\ P_A(\omega_m^{m-1}) \end{pmatrix}.$$

- Is M invertible? How do we find M^{-1}?

Claim

The inverse of matrix M is found by simply changing the signs of the exponents and dividing by m.

Let

$$Q = \frac{1}{m}\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_m^{-1} & \omega_m^{-2} & \dots & \omega_m^{-(m-1)} \\ 1 & \omega_m^{-2} & \omega_m^{-2\cdot 2} & \dots & \omega_m^{-2\cdot(m-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_m^{-(m-1)} & \omega_m^{-2(m-1)} & \dots & \omega_m^{-(m-1)(m-1)} \end{pmatrix}.$$

We need to prove that $MQ = I(= QM)$, where I is the identity matrix.

When we multiply MQ, the (i,j) entry in the product matrix is given by the product of the ith row of M and the jth column of Q:

$$(MQ)_{i,j} = \frac{1}{m}\begin{pmatrix}1 & \omega_m^i & \omega_m^{2\cdot i} & \cdots & \omega_m^{i\cdot(m-1)}\end{pmatrix}\begin{pmatrix}1\\ \omega_m^{-j}\\ \omega_m^{-2j}\\ \vdots\\ \omega_m^{-(m-1)j}\end{pmatrix}$$

$$= \frac{1}{m}\sum_{k=0}^{m-1}\omega_m^{ik}\,\omega_m^{-jk} = \frac{1}{m}\sum_{k=0}^{m-1}\omega_m^{(i-j)k}$$

We have established that $(MQ)_{i,j} = \frac{1}{m}\sum_{k=0}^{m-1}\omega_m^{(i-j)k}$. We now have two possibilities:

1. if $i = j$, then

$$\begin{aligned}(MQ)_{i,j} &= \frac{1}{m}\sum_{k=0}^{m-1}\omega_m^0 \\ &= \frac{1}{m}\sum_{k=0}^{m-1}1 \\ &= \frac{1}{m}\cdot m = 1;\end{aligned}$$

2. if $i \neq j$, then $\sum_{k=0}^{m-1} \omega_m^{(i-j)k}$ represents the sum of a geometric progression with m terms and common ratio ω_m^{i-j}. Recalling that

$$1 + r + r^2 + \ldots + r^{m-1} = \frac{1 - r^m}{1 - r},$$

we have

$$\begin{aligned}(MQ)_{i,j} &= \frac{1}{m} \cdot \frac{1 - \omega_m^{(i-j)m}}{1 - \omega_m^{i-j}} \\ &= \frac{1}{m} \cdot \frac{1 - (\omega_m^m)^{i-j}}{1 - \omega_m^{i-j}} \\ &= \frac{1}{m} \cdot \frac{1 - 1}{1 - \omega_m^{i-j}} = 0.\end{aligned}$$

We have now proven that

$$(MQ)_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases},$$

that is, $MQ = I$. Therefore Q is the inverse of M, i.e.

$$M^{-1} = \frac{1}{m} \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_m^{-1} & \omega_m^{-2} & \dots & \omega_m^{-(m-1)} \\ 1 & \omega_m^{-2} & \omega_m^{-2\cdot 2} & \dots & \omega_m^{-2\cdot(m-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_m^{-(m-1)} & \omega_m^{-2(m-1)} & \dots & \omega_m^{-(m-1)(m-1)} \end{pmatrix}.$$

So we get

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ \vdots \\ A_{m-1} \end{pmatrix} = \frac{1}{m} \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_m^{-1} & \omega_m^{-2} & \dots & \omega_m^{-(m-1)} \\ 1 & \omega_m^{-2} & \omega_m^{-2\cdot 2} & \dots & \omega_m^{-2\cdot(m-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_m^{-(m-1)} & \omega_m^{-2(m-1)} & \dots & \omega_m^{-(m-1)(m-1)} \end{pmatrix} \begin{pmatrix} P_A(1) \\ P_A(\omega_m) \\ P_A(\omega_m^2) \\ \vdots \\ P_A(\omega_m^{m-1}) \end{pmatrix}.$$

This is not so different to the original matrix-vector product!

The 'reconstruct' step requires us to convert from the sequence of values

$$\langle P_A(1), P_A(\omega_m), P_A(\omega_m^2), \ldots, P_A(\omega_m^{m-1})\rangle,$$

which we denoted by

$$\langle \widehat{A}_0, \widehat{A}_1, \widehat{A}_2, \ldots, \widehat{A}_{m-1}\rangle$$

back to the sequence of coefficients

$$\langle A_0, A_1, A_2, \ldots, A_{m-1}\rangle,$$

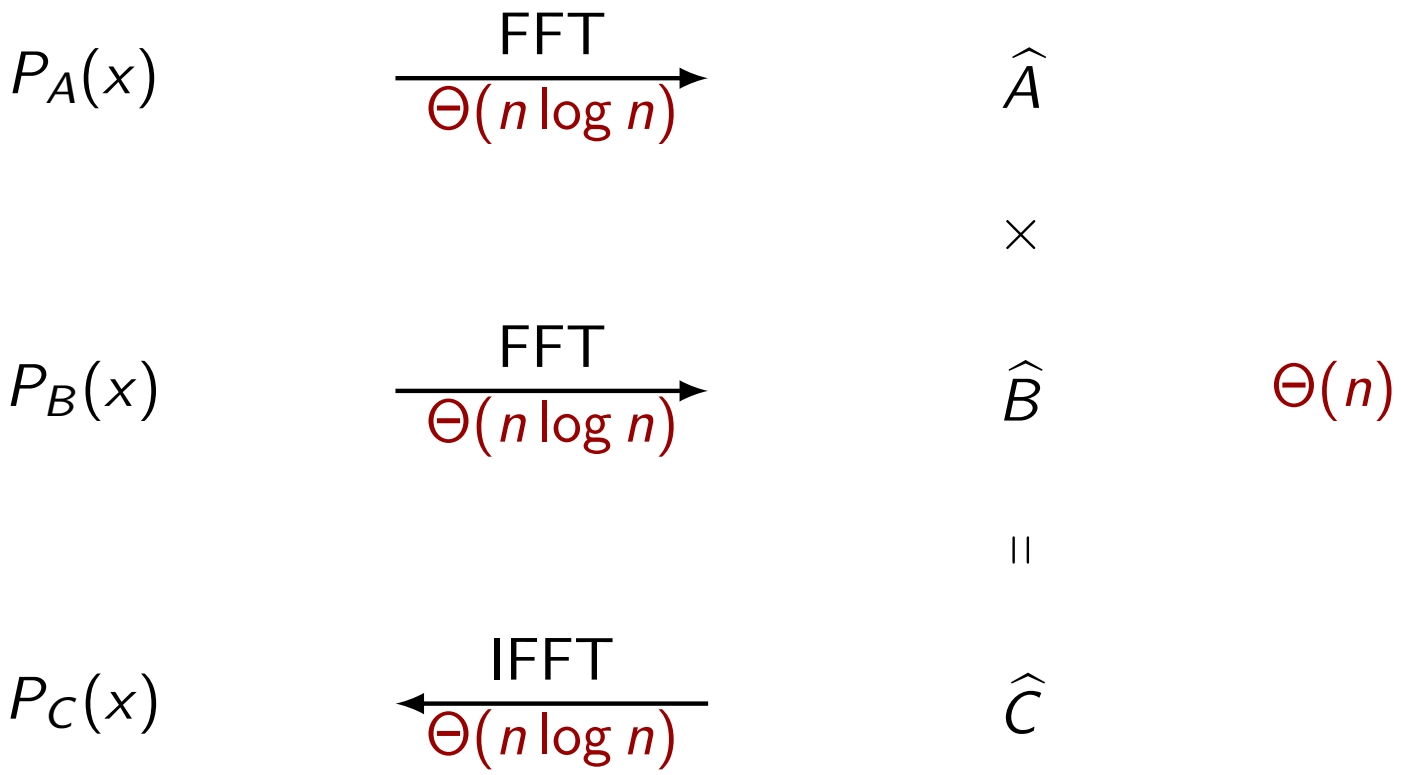
i.e. to compute the inverse DFT. We can use the same FFT algorithm with just two changes:

1. the root of unity ω_m is replaced by $\overline{\omega_m} = e^{-i\frac{2\pi}{m}}$,
2. the resulting output values are divided by m.

- Computer science books take the forward DFT operation to be the evaluation of the corresponding polynomial at all roots of unity $\omega_m^k = \exp(2\pi k\, i/m)$ and the inverse DFT to be the evaluation of the polynomial at the complex conjugates of the roots of unity, i.e. at $\omega_m^{-k} = \exp(-2\pi k\, i/m)$.

- However, electrical engineering books do the opposite, with the direct DFT evaluating the polynomial at ω_m^{-k} and the inverse DFT at ω_m^k.

- While for the purpose of multiplying polynomials both choices are equally good, the choice made by the electrical engineers is otherwise better. This will be explained in COMP4121 *Advanced Algorithms* in the context of JPEG compression.

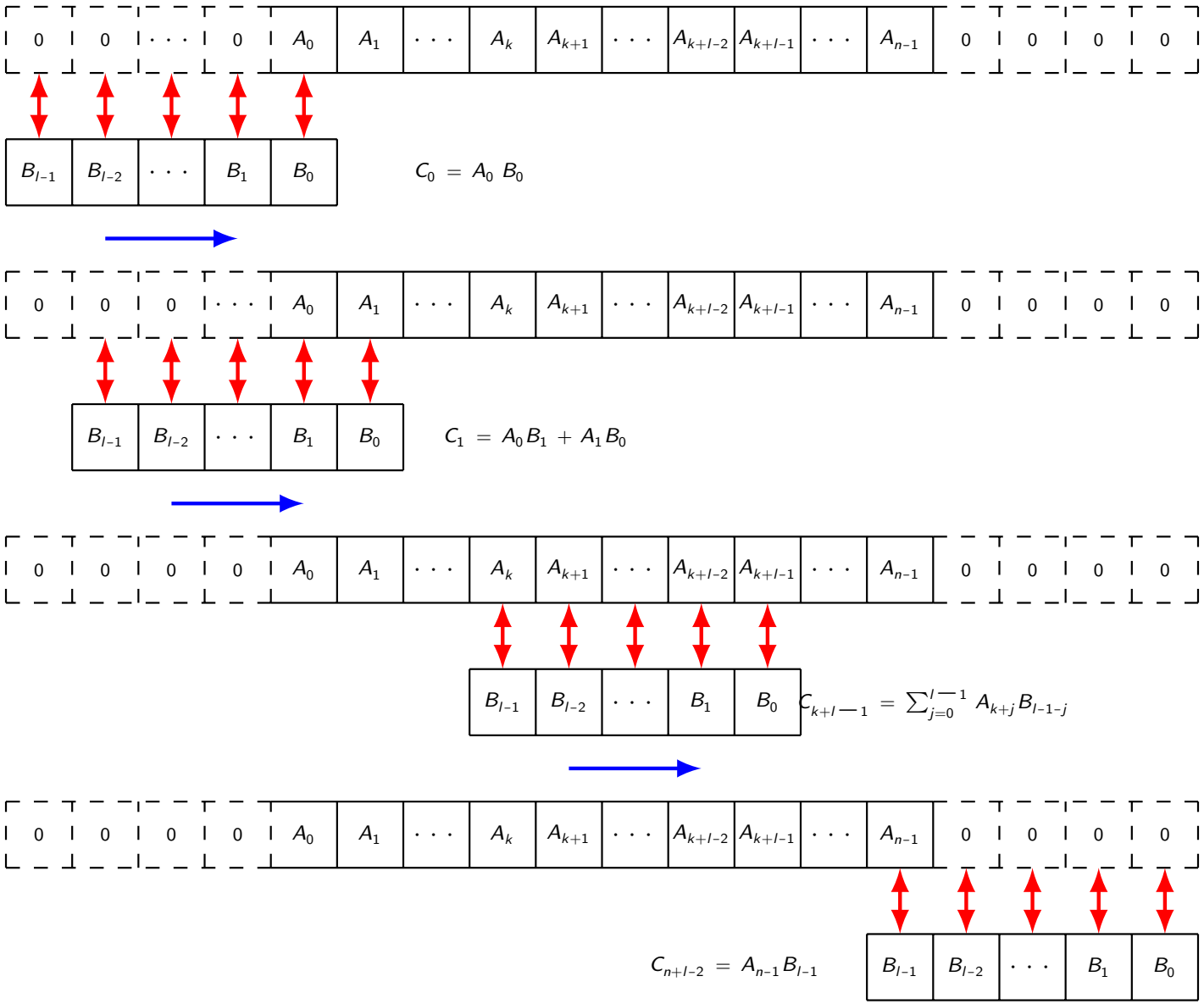

$P_A(x)$
FFT
$\Theta(n \log n)$
$\widehat{A}$
$\times$
$P_B(x)$
FFT
$\Theta(n \log n)$
$\widehat{B}$
$\Theta(n)$
$=$
$P_C(x)$
IFFT
$\Theta(n \log n)$
$\widehat{C}$

- We can now compute the product $P_C(x) = P_A(x)\,P_B(x)$ of two polynomials $P_A(x)$ and $P_B(x)$ in time $\Theta(n \log n)$.
- Recall that the coefficients of $P_C(x)$ are given by

$$C_t = \sum_{i+j=t} A_i B_j.$$

  The coefficient sequence C is the convolution of the sequences A and B, denoted $A \star B$.
- So we can find the convolution of two n-term sequences in $\Theta(n \log n)$.
- Indeed, none of this requires that the two sequences be of the same length!

Visualizing Convolution $C = A \star B$



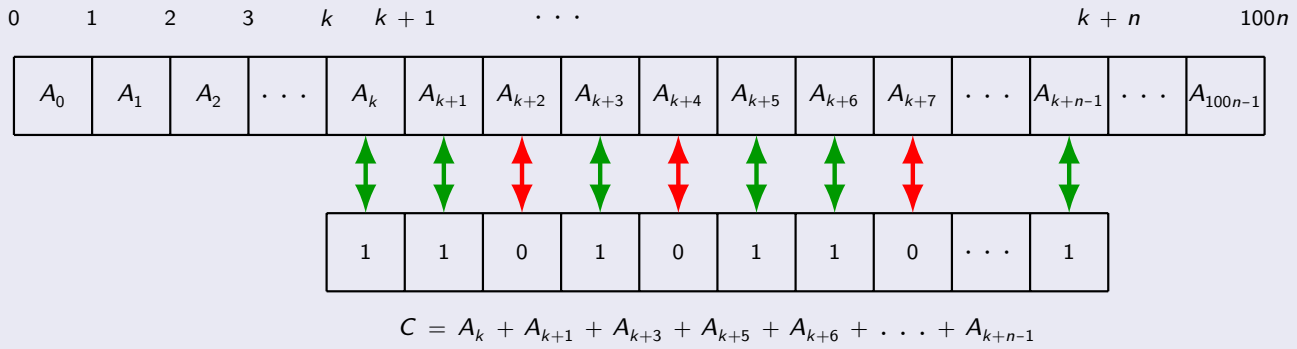
Problem

Suppose you are given a map of a straight sea shore of length $100n$ meters as a sequence on $100n$ numbers such that A_i is the number of fish between i^{th} meter of the shore and $(i+1)^{th}$ meter, $0 \leq i \leq 100n - 1$.

You also have a net of length n meters but unfortunately it has holes in it. The net is described as a sequence N of n ones and zeros, where 0's denote where the holes are.

If you throw the net starting at meter k and ending at meter $k + n$, then you will catch only the fish in one meter stretches of the shore where the corresponding bit of the net is 1.

Problem (continued)



Find the spot where you should place the left end of your net in order to catch the largest possible number of fish using an algorithm which runs in time $O(n \log n)$.

Hint

Let N' be the net sequence N in the reverse order. Compute $A \star N'$ and look for the peak of that sequence.

Problem

Five pirates have to split 100 bars of gold. They all line up and proceed as follows:

- The first pirate in line gets to propose a way to split up the gold (for example: everyone gets 20 bars).
- The pirates, including the one who proposed, vote on whether to accept the proposal. If the proposal is rejected, the pirate who made the proposal is killed.
- The next pirate in line then makes his proposal, and the 4 pirates vote again. If the vote is tied (2 vs 2) then the proposing pirate is still killed. Only majority can accept a proposal.

This process continues until a proposal is accepted or there is only one pirate left.

Problem (continued)

Assume that every pirate has the same priorities, in the following order:

1. not having to walk the plank;
2. getting as much gold as possible;
3. seeing other pirates walk the plank, just for fun.

Problem (continued)

What proposal should the first pirate make?

Hint

Assume first that there are only two pirates, and see what happens. Then assume that there are three pirates and that they have figured out what happens if there were only two pirates and try to see what they would do. Further, assume that there are four pirates and that they have figured out what happens if there were only three pirates, try to see what they would do. Finally assume there are five pirates and that they have figured out what happens if there were only four pirates.

That's All, Folks!!