CS 4104: Data and Algorithm Analysis

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Fall 2010

Changing the Model (1)

What if we settle for the “approximate best?”

Types of guarentees, given that the algorithm produces X and the best is Y:

1. $X = Y$. [Deterministic algorithm]
2. X’s rank is “close to” Y’s rank: [Approximation]

$$rank(X) \leq rank(Y) + \text{“small”}.$$

3. X is “usually” Y. [Probabilistic]

$$\mathbf{P}(X = Y) \geq \text{“large”}.$$

4. X’s rank is “usually” “close” to Y’s rank. [Heuristic]

Changing the Model (2)

We can also sacrifice reliability for speed:

1. We find the best, “usually” fast.
2. We find the best fast, or we don’t get an answer at all (but fast).

Examples for Findmax

Choose m elements at random, and pick the best.

- For large n, if $m = \log n$, the answer is pretty good.
- Cost is $m - 1$.
- Rank is $\frac{mn}{m+1}$.

Probabilistic Algorithms

Probabilistic algorithms include steps that are affected by **random** events.

Problem: Pick one number in the upper half of the values in a set.

1. Pick maximum: $n - 1$ comparisons.
2. Pick maximum from just over 1/2 of the elements: $n/2$ comparisons.

Can we do better? Not if we want a **guarantee**.

Probabilistic Algorithm

Pick 2 numbers and choose the greater.

This will be in the upper half with probability 3/4.

Not good enough? Pick more numbers!

For k numbers, greatest is in upper half with probability $1 - 2^{-k}$.

Monte Carlo Algorithm: Good running time, result not guaranteed.

Las Vegas Algorithm: Result guaranteed, but not running time.

Sorting

Initial model:

- Sort key has a linear order (comparable).
- We have an array of elements.
- We wish to sort the elements in the array.
- We get information about elements only by comparison of two elements.
- We can preserve order information only by swapping a pair of elements.

Sorting

Initial model:

- Sort key has a linear order (comparable).
- We have an array of elements.
- We wish to sort the elements in the array.
- We get information about elements only by comparison of two elements.
- We can preserve order information only by swapping a pair of elements.

To simplify analysis:

- Assume all elements are unique.
- For analysis purposes, consider the input to be a permutation of the values 1 to n.

Sorting

Initial model:

- Sort key has a linear order (comparable).
- We have an array of elements.
- We wish to sort the elements in the array.
- We get information about elements only by comparison of two elements.
- We can preserve order information only by swapping a pair of elements.

To simplify analysis:

- Assume all elements are unique.
- For analysis purposes, consider the input to be a permutation of the values 1 to n.

What if the ALGORITHM could make this assumption?

Swap Sorts (1)

Repeatedly scan input, swapping any out-of-order elements.

Bubble sort: $O(n^2)$ in worst case.

Inversions of an element: the number of smaller elements to the right of the element.

The sum of inversions for all elements is the number of swaps required by bubblesort.

ANY algorithm that removes one inversion per swap requires at least this many swaps.

Swap Sorts (2)

Worst number of inversions:

Best number of inversions:

Average number of inversions:

- Note that the sum of the total inversions for any permutation and its reverse is $\frac{n(n-1)}{2}$.
- Alternative view: every one of the $\frac{n(n-1)}{2}$ possible inversions occurs in a given permutation or its reverse.

Heap Sort (1)

Heap: complete binary tree with the value of any node at least as large as its two children.

Algorithm:

- Build the heap.
- Repeat *n* times:
  - Remove the root.
  - Repair the heap.

This gives us list in reverse sorted order.

Since the heap is a complete binary tree, it can be stored in an array.

Heap Sort (2)

To delete max element:

- Swap the last element in the heap with the first (root).
- Repeatedly swap the placeholder with larger of its two children until done.

Building the heap

To build a heap, first heapify the two subheaps, then push down the root to its proper position.

- Cost: $f(n) \leq 2f(n/2) + 2\log n$.

Alternatively: Start at first internal node and, moving up the array, siftdown each element.

- Cost:

$$\begin{aligned} f(n) &= \sum_{i=1}^{\log n} (i-1)\frac{n}{2^i} \\ &= \frac{n}{2}\sum_{i=1}^{\log n - 1} \frac{i}{2^i} < 2\frac{n}{2} = n. \end{aligned}$$

Quicksort

Algorithm:

- Pick a pivot value.
- Split the array into elements less than the pivot and elements greater than the pivot.
- Recursively sort the sublists.

Worst case:

Pick the pivot at random, so that no particular input has bad performance.

Quicksort Average Cost (1)

$$f(n) = \begin{cases} 0 & n \leq 1 \\ n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} (f(i) + f(n-i-1)) & n > 1 \end{cases}$$

Since the two halves of the summation are identical,

$$f(n) = \begin{cases} 0 & n \leq 1 \\ n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} f(i) & n > 1 \end{cases}$$

Multiplying both sides by n yields

$$nf(n) = n(n-1) + 2 \sum_{i=0}^{n-1} f(i).$$

Average Cost (2)

Get rid of the full history by subtracting $nf(n)$ from $(n+1)f(n+1)$

$$
\begin{aligned}
nf(n) &= n(n-1) + 2\sum_{i=1}^{n-1} f(i) \\
(n+1)f(n+1) &= (n+1)n + 2\sum_{i=1}^{n} f(i) \\
(n+1)f(n+1) - nf(n) &= 2n + 2f(n) \\
(n+1)f(n+1) &= 2n + (n+2)f(n) \\
f(n+1) &= \frac{2n}{n+1} + \frac{n+2}{n+1} f(n).
\end{aligned}
$$

Average Cost (3)

Note that $\frac{2n}{n+1} \leq 2$ for $n \geq 1$. Expanding the recurrence, we get

$$\begin{aligned}
f(n+1) &\leq 2 + \frac{n+2}{n+1} f(n) \\
&= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n} f(n-1)\right) \\
&= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n}\left(2 + \frac{n}{n-1} f(n-2)\right)\right) \\
&= 2 + \frac{n+2}{n+1}\left(2 + \cdots + \frac{4}{3}\left(2 + \frac{3}{2} f(1)\right)\right)
\end{aligned}$$

Average Cost (3)

$$
\begin{aligned}
&= 2\left(1 + \frac{n+2}{n+1} + \frac{n+2}{n+1}\frac{n+1}{n} + \cdots \right. \\
&\qquad \left. + \frac{n+2}{n+1}\frac{n+1}{n}\cdots\frac{3}{2}\right) \\
&= 2\left(1 + (n+2)\left(\frac{1}{n+1} + \frac{1}{n} + \cdots + \frac{1}{2}\right)\right) \\
&= 2 + 2(n+2)\left(\mathcal{H}_{n+1} - 1\right) \\
&= \Theta(n \log n).
\end{aligned}
$$

Lower Bound for Sorting (1)

What is the smallest number of comparisons needed to sort n values?

Clearly, sorting is as hard as finding the min and max element: $\lceil 3n/2 \rceil - 2$.

- Why?

Information theory says that, if an algorithm uses only binary decisions to distinguish between n possibilities, then it must use at least $\log n$ such decisions on average.

How is this relevant?

Lower Bound for Sorting (2)

There are $n!$ permutations to the input array.

So, by information theory, we need at least $\log n! = \Theta(n \log n)$ comparisons.

Using the decision tree model, what is the average depth of a node?

This is also $\Theta(\log n!)$.

Linear Insert Sort

Put the element i into a sorted list of the first $i - 1$ elements.

Worst case cost:

Best case cost:

Average case cost:

What if we use binary search? (This is called binary insert sort.)

Optimal Sorting (1)

If we count ONLY comparisons, binary insert sort is pretty good.

What is the absolute minimum number of comparisons needed to sort?

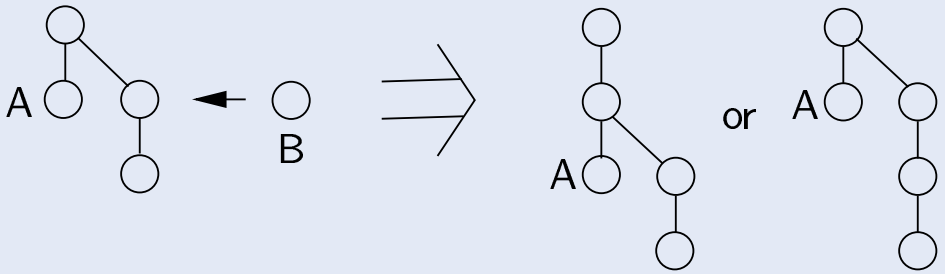
For $n = 5$, how many comparisons do we need for binary insert sort?

Binary search is best for what values of n?

Binary search is worst for what values of n?

Optimal Sorting (2)

Build the following poset:



Now, put in the fifth element (B) into the chain of 3.

Now, put in the off-element (A).

Total cost?

Ten Elements

Pair the elements: 5 comparisons.

Sort the winners of the pairings, using the previous algorithm: 7 comparisons.

Now, all we need to do is to deal with the original losers.

General algorithm:

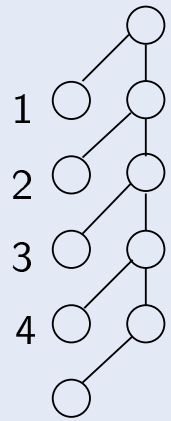
- Pair up all the nodes with $\lfloor \frac{n}{2} \rfloor$ comparisons.
- Recursively sort the winners.
- Fold in the losers.

Finishing the Sort (1)

We will use binary insert to place the losers.

However, we are free to choose best ordering for inserting.

Recall that binary search is best for $2^k - 1$ items.



Finishing the Sort (2)

Pick the order of inserts to optimize the binary searches.

- 3 (2 compares: size 3)
- 4 (2 compares: size either 2 or 3, depending on where element 3 ends up)
- 1 (3 compares: size between 5 and 7)
- 2 (3 compares: size between 5 and 7)

We can form an algorithm: Binary Merge.

This sort is called **merge insert sort**

Optimal Sort Algorithm?

- Merge insert sort is pretty good, but is it optimal?
- It does not match the information theoretic lower bound for $n = 12$.
  - Merge insert sort gives 30 instead of 29 comparison.
- BUT, exhaustive search shows the information theoretic bound is an underestimate for $n = 12$. 30 is best.
- Call the optimal worst cost for n elements $S(n)$.
  - $S(n+1) \leq S(n) + \lceil \log(n+1) \rceil$.
    Otherwise, we would sort n elements and binary insert the last.
  - For all n and m, $S(n+m) \leq S(n) + S(m) + M(m,n)$ for $M(m,n)$ the best time to merge two sorted lists.
  - For $n = 47$, we can do better by splitting into pieces of size 5 and 42, then merging.

A Truly Optimal Algorithm

Pick the best set of comparisons for size 2.

Then for size 3, 4, 5, ...

Combine them together into one program with a big case statement.

Is this an algorithm?

Numbers

Examples of problems:

- Raise a number to a power.
- Find common factors for two numbers.
- Tell whether a number is prime.
- Generate a random integer.
- Multiply two integers.

These operations use all the digits, and cannot use floating point approximation.

For large numbers, cannot rely on hardware (constant time) operations.

- Measure input size by number of binary digits.
- Multiply, divide become expensive.

Analysis of Number Problems

Analysis problem: Cost may depend on properties of the number other than size.

- It is easy to check an even number for primeness.

Considering cost over all k-bit inputs, cost grows with k.

Features:

- Arithmetical operations are not cheap.
- There is only one instance of value n.
- There are 2^k instances of length k or less.
- The size (length) of value n is $\log n$.
- The cost may decrease when n increases in value, but generally increases when n increases in size (length).

Exponentiation (1)

How do we compute m^n?

We could multiply $n - 1$ times.
Can we do better?

Approaches to divide and conquer:

- Relate m^n to k^n for $k < m$.
- Relate m^n to m^k for $k < n$.

If n is even, then $m^n = m^{n/2} m^{n/2}$.

If n is odd, then $m^n = m^{\lfloor n/2 \rfloor} m^{\lfloor n/2 \rfloor} m$.

Exponentiation (2)

```
int Power(int base, int exp) {
  int half, total;
  if exp = 0 return 1;
  half = Power(base, exp/2);
  total = half * half;
  if (odd(exp)) then total = total * base;
  return total;
}
```

Analysis of Power

$$f(n) = \begin{cases} 0 & n = 1 \\ f(\lfloor n/2 \rfloor) + 1 + n \bmod 2 & n > 1 \end{cases}$$

Solution: $f(n) = \lfloor \log n \rfloor + \beta(n) - 1$
where β is the number of 1's in binary representation of n.

How does this cost compare with the problem size?

Is this the best possible? What if $n = 15$?

What if n stays the same but m changes over many runs?

In general, finding the best set of multiplications is expensive (probably exponential).

Largest Common Factor (1)

The largest common factor of two numbers is the largest integer that divides both evenly.

Observation: If k divides n and m, then k divides $n - m$.

So, $f(n, m) = f(n - m, n) = f(m, n - m) = f(m, n)$.

Observation: There exists k and l such that

$$n = km + l \text{ where } m > l \geq 0.$$

$$n = \lfloor n/m \rfloor m + n \bmod m.$$

So, $f(n, m) = f(m, l) = f(m, n \bmod m)$.

Largest Common Factor (2)

$$f(n,m) = \begin{cases} n & m = 0 \\ f(m, n \bmod m) & m > 0 \end{cases}$$

```
int LCF(int n, int m) {
  if (m == 0) return n;
  return LCF(m, n % m);
}
```

Analysis of LCF

How big is n mod m relative to n?

$$\begin{aligned} n \geq m &\Rightarrow n/m \geq 1 \\ &\Rightarrow 2\lfloor n/m \rfloor > n/m \\ &\Rightarrow m\lfloor n/m \rfloor > n/2 \\ &\Rightarrow n - n/2 > n - m\lfloor n/m \rfloor = n \bmod m \\ &\Rightarrow n/2 > n \bmod m \end{aligned}$$

The first argument must be halved in no more than 2 iterations.

Total cost:

Matrix Multiplication

Given: $n \times n$ matrices A and B.

Compute: $C = A \times B$.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Straightforward algorithm:

- $\Theta(n^3)$ multiplications and additions.

Lower bound for any matrix multiplication algorithm: $\Omega(n^2)$.

Another Approach

Compute:

$$
\begin{aligned}
m_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\
m_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
m_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\
m_4 &= (a_{11} + a_{12})b_{22} \\
m_5 &= a_{11}(b_{12} - b_{22}) \\
m_6 &= a_{22}(b_{21} - b_{11}) \\
m_7 &= (a_{21} + a_{22})b_{11}
\end{aligned}
$$

Then:

$$
\begin{aligned}
c_{11} &= m_1 + m_2 - m_4 + m_6 \\
c_{12} &= m_4 + m_5 \\
c_{21} &= m_6 + m_7 \\
c_{22} &= m_2 - m_3 + m_5 - m_7
\end{aligned}
$$

7 multiplications and 18 additions/subtractions.

Strassen's Algorithm (1)

(1) Trade more additions/subtractions for fewer multiplications in 2×2 case.

(2) Divide and conquer.

In the straightforward implementation, 2×2 case is:

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}$$
$$c_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$c_{21} = a_{21}b_{11} + a_{22}b_{21}$$
$$c_{22} = a_{21}b_{12} + a_{22}b_{22}$$

Requires 8 multiplications and 4 additions.

Strassen's Algorithm (2)

Divide and conquer step:

Assume n is a power of 2.

Express $C = A \times B$ in terms of $\frac{n}{2} \times \frac{n}{2}$ matrices.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Strassen's Algorithm (3)

By Strassen's algorithm, this can be computed with 7 multiplications and 18 additions/subtractions of $n/2 \times n/2$ matrices.

Recurrence:

$$\begin{aligned} T(n) &= 7T(n/2) + 18(n/2)^2 \\ T(n) &= \Theta(n^{\log_2 7}) = \Theta(n^{2.81}). \end{aligned}$$

Current "fastest" algorithm is $\Theta(n^{2.376})$

Open question: Can matrix multiplication be done in $O(n^2)$ time?

Divide and Conquer Recurrences (1)

These have the form:

$$\begin{aligned} T(n) &= aT(n/b) + cn^k \\ T(1) &= c \end{aligned}$$

... where a, b, c, k are constants.

A problem of size n is divided into a subproblems of size n/b, while cn^k is the amount of work needed to combine the solutions.

Divide and Conquer Recurrences (2)

Expand the sum; assume $n = b^m$.

$$\begin{aligned} T(n) &= a(aT(n/b^2) + c(n/b)^k) + cn^k \\ &= a^m T(1) + a^{m-1} c(n/b^{m-1})^k + \cdots + ac(n/b)^k + cn^k \\ &= ca^m \sum_{i=0}^{m} (b^k/a)^i \end{aligned}$$

$a^m = a^{\log_b n} = n^{\log_b a}$

The summation is a geometric series whose sum depends on the ratio

$$r = b^k/a.$$

There are 3 cases.

D & C Recurrences (3)

(1) $r < 1$

$$\sum_{i=0}^{m} r^i < 1/(1-r), \qquad \text{a constant.}$$

$$T(n) = \Theta(a^m) = \Theta(n^{\log_b a}).$$

(2) $r = 1$

$$\sum_{i=0}^{m} r^i = m + 1 = \log_b n + 1$$

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^k \log n)$$

D & C Recurrences (4)

(3) $r > 1$

$$\sum_{i=0}^{m} r^i = \frac{r^{m+1} - 1}{r - 1} = \Theta(r^m)$$

So, from $T(n) = ca^m \sum r^i$,

$$\begin{aligned} T(n) &= \Theta(a^m r^m) \\ &= \Theta(a^m (b^k/a)^m) \\ &= \Theta(b^{km}) \\ &= \Theta(n^k) \end{aligned}$$

Summary

Theorem 3.4:

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

Apply the theorem:
$T(n) = 3T(n/5) + 8n^2$.
$a = 3, b = 5, c = 8, k = 2$.
$b^k/a = 25/3$.

Case (3) holds: $T(n) = \Theta(n^2)$.

Prime Numbers

How do we tell if a number is prime?

One approach is the prime sieve: Test all prime up to $\lfloor \sqrt{n} \rfloor$.

This requires up to $\lfloor \sqrt{n} \rfloor - 1$ divisions.

- How does this compare to the input size?

Note that it is easy to check the number of times 2 divides n for the binary representation

- What about 3?
- What if n is represented in trinary?

Is there a polynomial time algorithm?

Facts about Primes

Some useful theorems from Number Theory:

- **Prime Number Theorem**: The number of primes less than n is (approximately)

$$\frac{n}{\ln n}$$

  - The average distance between primes is $\ln n$.
- **Prime Factors Distribution Theorem**: For large n, on average, n has about $\ln \ln n$ different prime factors with a standard deviation of $\sqrt{\ln \ln n}$.

To prove that a number is composite, need only one factor.
What does it take to prove that a number is prime?
Do we need to check all $\sqrt{n}$ candidates?

Probablistic Algorithms

Some probablistic algorithms:

- Prime(n) $=$ FALSE.
- With probability $1/\ln n$, Prime(n) $=$ TRUE.
- Pick a number m between 2 and $\sqrt{n}$. Say n is prime iff m does not divide n.

Using number theory, can create cheap test that determines a number to be composite (if it is) 50% of the time.

```
Prime(n) {
  for(i=0; i<COMFORT; i++)
    if !CHEAPTEST(n)
      return FALSE;
  return TRUE;
}
```

Of course, this does nothing to help you find the factors!

Random Numbers

Which sequences are random?

- 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
- 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
- 2, 7, 1, 8, 2, 8, 1, 8, 2, ...

Meanings of “random”:

- Cannot predict the next item: **unpredictable**.
- Series cannot be described more briefly than to reproduce it: **equidistribution**.

There is no such thing as a random number sequence, only “random enough” sequences.

A sequence is **pseudorandom** if no future term can be predicted in polynomial time, given all past terms.

A Good Random Number Generator

Most computer systems use a deterministic algorithm to select pseudorandom numbers.

Linear congruential method:

- Pick a **seed** $r(1)$. Then,

$$r(i) = (r(i-1) \times b) \bmod t.$$

Resulting numbers must be in range: What happens if

$r(i) = r(j)$?

Must pick good values for b and t.

- t should be prime.

Random Number examples

$r(i) = 6r(i-1) \bmod 13 =$

$..., 1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1, ...$

$r(i) = 7r(i-1) \bmod 13 =$

$..., 1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1, ...$

$r(i) = 5r(i-1) \bmod 13 =$

$..., 1, 5, 12, 8, 1, ...$

$..., 2, 10, 11, 3, 2, ...$

$..., 4, 7, 9, 6, 4, ...$

$..., 0, 0, ...$

Suggested generator: $r(i) = 16807r(i-1) \bmod 2^{31} - 1$.

Introduction to the Sliderule

Compared to addition, multiplication is hard.

In the physical world, addition is merely concatenating two lengths.

Observation:

$$\log nm = \log n + \log m.$$

Therefore,

$$nm = \text{antilog}(\log n + \log m).$$

What if taking logs and antilogs were easy?

Introduction to the Sliderule (2)

The sliderule does exactly this!

- It is essentially two rulers in log scale.
- Slide the scales to add the lengths of the two numbers (in log form).
- The third scale shows the value for the total length.

Representing Polynomials

A vector **a** of n values can uniquely represent a polynomial of degree $n - 1$

$$P_{\mathbf{a}}(x) = \sum_{i=0}^{n-1} \mathbf{a}_i x^i.$$

Alternatively, a degree $n - 1$ polynomial can be uniquely represented by a list of its values at n distinct points.

- Finding the value for a polynomial at a given point is called **evaluation**.
- Finding the coefficients for the polynomial given the values at n points is called **interpolation**.

Multiplication of Polynomials

To multiply two $n-1$-degree polynomials A and B normally takes $\Theta(n^2)$ coefficient multiplications.

However, if we evaluate both polynomials, we can simply multiply the corresponding pairs of values to get the values of polynomial AB.

Process:

- Evaluate polynomials A and B at enough points.
- Pairwise multiplications of resulting values.
- Interpolation of resulting values.

Multiplication of Polynomials (2)

This can be faster than $\Theta(n^2)$ IF a fast way can be found to do evaluation/interpolation of $2n - 1$ points (normally this takes $\Theta(n^2)$ time).

Note that evaluating a polynomial at 0 is easy, and that if we evaluate at 1 and -1, we can share a lot of the work between the two evaluations.

Can we find enough such points to make the process cheap?

An Example

Polynomial A: $x^2 + 1$.
Polynomial B: $2x^2 - x + 1$.
Polynomial AB: $2x^4 - x^3 + 3x^2 - x + 1$.

Notice:

$$\begin{aligned} AB(-1) &= (2)(4) = 8 \\ AB(0) &= (1)(1) = 1 \\ AB(1) &= (2)(2) = 4 \end{aligned}$$

But: We need 5 points to nail down Polynomial AB. And, we also need to interpolate the 5 values to get the coefficients back.

Nth Root of Unity

The key to fast polynomial multiplication is finding the right points to use for evaluation/interpolation to make the process efficient.

Complex number ω is a **primitive nth root of unity** if

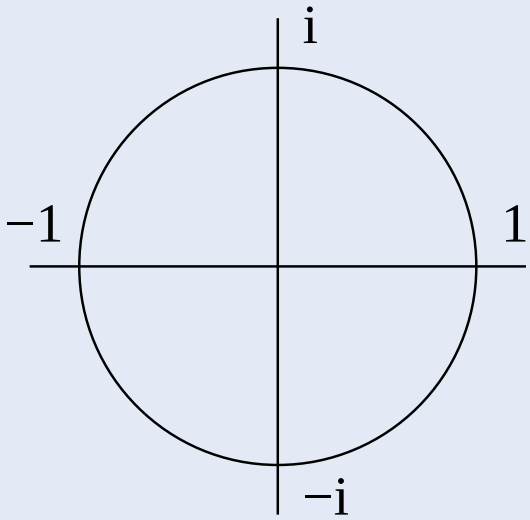
1. $\omega^n = 1$ and
2. $\omega^k \neq 1$ for $0 < k < n$.

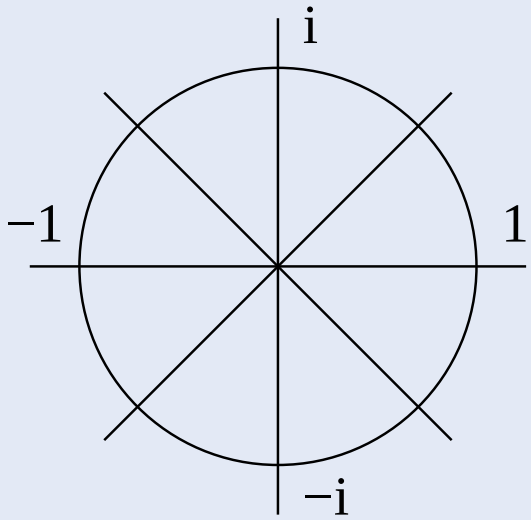
$\omega^0, \omega^1, ..., \omega^{n-1}$ are the **nth roots of unity**.

Example:

- For $n = 4$, $\omega = i$ or $\omega = -i$.

Nth Root of Unity (cont)





$n = 4, \omega = i.$
$n = 8, \omega = \sqrt{i}.$

Discrete Fourier Transform

Define an $n \times n$ matrix $V(\omega)$ with row i and column j as

$$V(\omega) = (\omega^{ij}).$$

Example: $n = 4$, $\omega = i$:

$$V(\omega) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

Let $\overline{a} = [a_0, a_1, ..., a_{n-1}]^T$ be a vector.
The **Discrete Fourier Transform** (DFT) of $\overline{a}$ is:

$$F_\omega = V(\omega)\overline{a} = \overline{v}.$$

This is equivalent to evaluating the polynomial at the nth roots of unity.

Array example

For $n = 8$, $\omega = \sqrt{i}$, $V(\omega) =$

$$\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & \sqrt{i} & i & i\sqrt{i} & -1 & -\sqrt{i} & -i & -i\sqrt{i} \\
1 & i & -1 & -i & 1 & i & -1 & -i \\
1 & i\sqrt{i} & -i & \sqrt{i} & -1 & -i\sqrt{i} & i & -\sqrt{i} \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
1 & -\sqrt{i} & i & -i\sqrt{i} & -1 & \sqrt{i} & -i & i\sqrt{i} \\
1 & -i & -1 & i & 1 & -i & -1 & i \\
1 & -i\sqrt{i} & -i & -\sqrt{i} & -1 & i\sqrt{i} & i & \sqrt{i}
\end{array}$$

Inverse Fourier Transform

The inverse Fourier Transform to recover $\overline{a}$ from $\overline{v}$ is:

$$F_\omega^{-1} = \overline{a} = [V(\omega)]^{-1} \cdot \overline{v}.$$

$$[V(\omega)]^{-1} = \frac{1}{n} V(\frac{1}{\omega}).$$

This is equivalent to interpolating the polynomial at the nth roots of unity.

An efficient divide and conquer algorithm can perform both the DFT and its inverse in $\Theta(n \lg n)$ time.

Fast Polynomial Multiplication

Polynomial multiplication of A and B:

- Represent an $n-1$-degree polynomial as $2n-1$ coefficients:

$$[a_0, a_1, ..., a_{n-1}, 0, ..., 0]$$

- Perform DFT on representations for A and B.
- Pairwise multiply results to get $2n-1$ values.
- Perform inverse DFT on result to get $2n-1$ degree polynomial AB.

FFT Algorithm

```
FFT(n, a0, a1, ..., an-1, omega, var V);
Output: V[0..n-1] of output elements.
begin
  if n=1 then V[0] = a0;
  else
    FFT(n/2, a0, a2, ... an-2, omega^2, U);
    FFT(n/2, a1, a3, ... an-1, omega^2, W);
    for j=0 to n/2-1 do
      V[j] = U[j] + omega^j W[j];
      V[j+n/2] = U[j] - omega^j W[j];
end
```