CSE 548: (Design and) Analysis of Algorithms

Divide-and-conquer Algorithms

R. Sekar
## Divide-and-Conquer: A versatile strategy

<table>
<thead>
<tr>
<th>Steps</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Break a problem into subproblems that are smaller instances of the same problem</td>
<td><strong>Conceptual simplification</strong></td>
</tr>
<tr>
<td>Recursively solve these subproblems</td>
<td><strong>Speed up:</strong></td>
</tr>
<tr>
<td>Combine these answers to obtain the solution to the problem</td>
<td><em>rapidly (exponentially) reduce problem space</em></td>
</tr>
<tr>
<td></td>
<td><em>exploit commonalities in subproblem solutions</em></td>
</tr>
</tbody>
</table>

**Parallelism:** Divide-and-conquer algorithms are amenable to parallelization.

**Locality:** Their depth-first nature increases locality, extremely important for today’s processors.
Topics

1. Warmup
   - Overview
   - Search
   - H-Tree
   - Exponentiation

2. Sorting
   - Mergesort
   - Recurrences
   - Quicksort

3. Selection
   - Lower Bound
   - Radix sort
   - Select $k$-th min
   - Priority Queues

4. Closest pair

5. Multiplication
   - Matrix
   - Multiplication

6. FFT
   - Integer multiplication
   - Fourier Transform
   - DFT
   - FFT Algorithm
   - Fast multiplication
Binary Search

Problem: Find a key $k$ in an ordered collection

Examples: Sorted array $A[n]$: Compare $k$ with $A[n/2]$, then recursively search in $A[0 \cdots (n/2 - 1)]$ (if $k < A[n/2]$) or $A[n/2 \cdots n]$ (otherwise)

Binary search tree $T$: Compare $k$ with $\text{root}(T)$, based on the result, recursively search left or right subtree of root.

B-Tree: Hybrid of the above two. Root stores an array $M$ of $m$ keys, and has $m + 1$ children. Use binary search on $M$ to identify which child can contain $k$, recursively search that subtree.
H-tree: Planar embedding of full binary tree

Key properties
- fractal geometry — divide-and-conquer structure
- $n$ nodes in $O(n)$ area
- all root-to-leaf paths equal in length

Applications
- compact embedding of binary trees in VLSI
- hardware clock distribution
MkHtree($l, b, r, t, n$)

horizLine($\frac{b+t}{2}, l + \frac{l+r}{4}, r - \frac{l+r}{4}$)

vertLine($l + \frac{l+r}{4}, b + \frac{b+t}{4}, t - \frac{b+t}{4}$)

vertLine($r - \frac{l+r}{4}, b + \frac{b+t}{4}, t - \frac{b+t}{4}$)

if $n \leq 4$ return

MkHtree($l, \frac{b+t}{2}, \frac{l+r}{2}, t, \frac{n}{4}$)

MkHtree($\frac{l+r}{2}, \frac{b+t}{2}, r, t, \frac{n}{4}$)

MkHtree($l, b, \frac{l+r}{2}, \frac{b+t}{2}, \frac{n}{4}$)

MkHtree($\frac{l+r}{2}, b, r, \frac{b+t}{2}, \frac{n}{4}$)
Questions

- How compact is the embedding
- Ratio of minimum distance between nodes and the average area per node
- What is the root-to-leaf path length?
- Can we do better?
- Finally, how can we show that the algorithm is correct?

\[ \text{MkHtree}(l, b, r, t, n) \]

\[ \text{horizLine}(\frac{b+t}{2}, l + \frac{l+r}{4}, r - \frac{l+r}{4}) \]
\[ \text{vertLine}(l + \frac{l+r}{4}, b + \frac{b+t}{4}, t - \frac{b+t}{4}) \]
\[ \text{vertLine}(r - \frac{l+r}{4}, b + \frac{b+t}{4}, t - \frac{b+t}{4}) \]

\textbf{if} \ n \leq 4 \ \textbf{return}

\[ \text{MkHtree}(l, \frac{b+t}{2}, \frac{l+r}{2}, t, \frac{n}{4}) \]
\[ \text{MkHtree}(\frac{l+r}{2}, \frac{b+t}{2}, r, t, \frac{n}{4}) \]
\[ \text{MkHtree}(l, b, \frac{l+r}{2}, \frac{b+t}{2}, \frac{n}{4}) \]
\[ \text{MkHtree}(\frac{l+r}{2}, b, r, \frac{b+t}{2}, \frac{n}{4}) \]
Exponentiation

- How many multiplications are required to compute $x^n$?
- Can we use a divide-and-conquer approach to make it faster?

```plaintext
ExpBySquaring(n, x)

if $n > 1$

    $y = \text{ExpBySquaring}(\lfloor n/2 \rfloor, x^2)$

if $\text{odd}(n)$

    $y = x \times y$

return $y$

else return $x$
```
The ultimate divide-and-conquer algorithm is, of course, binary search: to find a key $k$ in a large file containing keys $z[0, 1, \ldots, n-1]$ in sorted order, we first compare $k$ with $z[n/2]$, and depending on the result we recurse either on the first half of the file, $z[0, \ldots, n/2-1]$, or on the second half, $z[n/2, \ldots, n-1]$. The recurrence now is $T(n) = T(\frac{n}{2}) + O(1)$, which is the case $a=1$, $b=2$, $d=0$. Plugging into our master theorem we get the familiar solution: a running time of just $O(\log n)$.

### 2.3 Mergesort

The problem of sorting a list of numbers lends itself immediately to a divide-and-conquer strategy: split the list into two halves, recursively sort each half, and then merge the two sorted sublists.

```python
function mergesort(a[1...n])
Input: An array of numbers a[1...n]
Output: A sorted version of this array
if n > 1:
    return merge(mergesort(a[1...n/2]), mergesort(a[n/2+1...n]))
else:
    return a
```

The correctness of this algorithm is self-evident, as long as a correct merge subroutine is specified. If we are given two sorted arrays $x[1...k]$ and $y[1...l]$, how do we efficiently merge them into a single sorted array $z[1...k+l]$? Well, the very first element of $z$ is either $x[1]$ or $y[1]$, whichever is smaller. The rest of $z[·]$ can then be constructed recursively.

```python
function merge(x[1...k], y[1...l])
if k = 0: return y[1...l]
if l = 0: return x[1...k]
if x[1] ≤ y[1]:
    return x[1] ◦ merge(x[2...k], y[1...l])
else:
    return y[1] ◦ merge(x[1...k], y[2...l])
```

Here ◦ denotes concatenation. This merge procedure does a constant amount of work per recursive call (provided the required array space is allocated in advance), for a total running time of $O(k+l)$. Thus merge's are linear, and the overall time taken by mergesort is $T(n) = 2T(n/2) + O(n)$, or $O(n \log n)$.
The ultimate divide-and-conquer algorithm is, of course, binary search: to find a key \( k \) in a large file containing keys \( z[0, 1, \ldots, n-1] \) in sorted order, we first compare \( k \) with \( z[n/2] \), and depending on the result we recurse either on the first half of the file, \( z[0, \ldots, n/2-1] \), or on the second half, \( z[n/2, \ldots, n-1] \). The recurrence now is
\[
T(n) = T(\frac{n}{2}) + O(1),
\]
which is the case \( a = 1, b = 2, d = 0 \). Plugging into our master theorem we get the familiar solution: a running time of just \( O(\log n) \).

### 2.3 Mergesort

The problem of sorting a list of numbers lends itself immediately to a divide-and-conquer strategy: split the list into two halves, recursively sort each half, and then merge the two sorted sublists.

```python
function mergesort(a[1..n])
Input: An array of numbers
Output: A sorted version of this array
if n > 1:
    return merge(mergesort(a[1..\lfloor n/2 \rfloor]), mergesort(a[\lceil n/2 \rceil + 1..n]))
else:
    return a
```

The correctness of this algorithm is self-evident, as long as a correct merge subroutine is specified. If we are given two sorted arrays \( x[1..k] \) and \( y[1..l] \), how do we efficiently merge them into a single sorted array \( z[1..k+l] \)? Well, the very first element of \( z \) is either \( x[1] \) or \( y[1] \), whichever is smaller. The rest of \( z[\cdot] \) can then be constructed recursively.

```python
function merge(x[1..k], y[1..l])
if k = 0:
    return y[1..l]
if l = 0:
    return x[1..k]
if x[1] \leq y[1]:
    return x[1] \circ merge(x[2..k], y[1..l])
else:
    return y[1] \circ merge(x[1..k], y[2..l])
```
Merge Sort Illustration

Figure 2.4 The sequence of merge operations in mergesort.

```
Input: 10 2 3 1 13 5 7 6
```

This viewpoint also suggests how mergesort might be made iterative. At any given moment, there is a set of “active” arrays—initially, the singletons—which are merged in pairs to give the next batch of active arrays. These arrays can be organized in a queue, and processed by repeatedly removing two arrays from the front of the queue, merging them, and putting the result at the end of the queue.

In the following pseudocode, the primitive operation inject adds an element to the end of the queue while eject removes and returns the element at the front of the queue.

```
function iterative-mergesort([a[1], ..., a[n]])
  Input: elements a₁, a₂, ..., aₙ to be sorted
  Q = [] (empty queue)
  for i = 1 to n:
    inject(Q,[a[i]])
  while |Q| > 1:
    inject(Q,merge(eject(Q),eject(Q)))
  return eject(Q)
```
Merge sort time complexity

- `mergesort(A)` makes two recursive invocations of itself, each with an array half the size of `A`.
- `merge(A, B)` takes time that is linear in `|A| + |B|`.
- Thus, the runtime is given by the recurrence:

\[ T(n) = 2T\left(\frac{n}{2}\right) + n \]

- In divide-and-conquer algorithms, we often encounter recurrences of the form:

\[ T(n) = aT\left(\frac{n}{b}\right) + O(n^d) \]

Can we solve them once for all?
Master Theorem

If $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$ for constants $a > 0$, $b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} 
O(n^d), & \text{if } d > \log_b a \\
O(n^d \log n), & \text{if } d = \log_b a \\
O(n^{\log_b a}), & \text{if } d < \log_b a 
\end{cases}$$
Proof of Master Theorem

Can be proved by induction, or by summing up the series where each term represents the work done at one level of this tree.
What if Master Theorem can’t be applied?

- Guess and check (prove by induction)
  - expand recursion for a few steps to make a guess
  - in principle, can be applied to any recurrence

- Akra-Bazzi method (not covered in class)
  - recurrences can be much more complex than that of Master theorem
QuickSort

**qs**(A, l, h) /*sorts A[l...h]*/

if \ l >= h return;

(h₁, l₂) = partition(A, l, h);

qs(A, l, h₁);
qs(A, l₂, h)

**partition**(A, l, h)

k = selectPivot(A, l, h); p = A[k];
swap(A, h, k);
i = l - 1; j = h;

while true do
  do i++ while A[i] < p;
  do j-- while A[j] > p;
  if i \geq j break;
  swap(A, i, j);

swap(A, i, h)

return (j, i + 1)
Analysis of Runtime of \textit{qs}

General case: Given by the recurrence $T(n) = n + T(n_1) + T(n_2)$

where $n_1$ and $n_2$ are the sizes of the two sub-arrays after partition.

Best case: $n_1 = n_2 = n/2$. By master theorem, $T(n) = O(n \log n)$

Worst case: $n_1 = 1, n_2 = n - 1$. By master theorem, $T(n) = O(n^2)$

- A fixed choice of pivot index, say, $h$, leads to worst-case behavior in common cases, e.g., input is sorted.

Lucky/unlucky split: Alternate between best- and worst-case splits.

\[
T(n) = n + T(1) + T(n-1) + n \quad \text{(worst case split)}
\]

\[
= n + 1 + (n-1) + 2T((n-1)/2) = 2n + 2T((n - 1)/2)
\]

which has an $O(n \log n)$ solution.

Three-fourths split:

\[
T(n) = n + T(0.25n) + T(0.75n) \leq n + 2T(0.75n) = O(n \log n)
\]
Average case analysis of *qs*

Define input distribution: All permutations equally likely

- Note: in always true, pivot selection does not matter!

Simplifying assumption: all elements are distinct. (Nonessential assumption)

Set up the recurrence: When all permutations are equally likely, the selected pivot has an equal chance of ending up at the *i*\(^{th}\) position in the sorted order, for all 1 ≤ *i* ≤ *n*. Thus, we have the following recurrence for the average case:

\[
T(n) = n + \frac{1}{n} \sum_{i=1}^{n-1} (T(i) + T(n - i))
\]

Solve recurrence: Cannot apply the master theorem, but since it seems that we get an *O*(*n* log *n*) bound even in seemingly bad cases, we can try to establish a *cn* log *n* bound via induction.
Establishing average case of \( qs \)

- Establish base case. (Trivial.)

- Induction step involves summation of the form \( \sum_{i=1}^{n-1} i \log i \).

  **Attempt 1:** bound \( \log i \) above by \( \log n \). (Induction fails.)

  **Attempt 2:** split the sum into two parts:

  \[
  \sum_{i=1}^{n/2} i \log i + \sum_{i=n/2+1}^{n-1} i \log i
  \]

  and apply the approx. to each half. (Succeeds with \( c \geq 4 \).)

  **Attempt 3:** replace the summation with the upper bound

  \[
  \int_{x=1}^{n} x \log x = \frac{x^2}{2} \left( \log x - \frac{1}{2} \right) \bigg|_{x=1}^{n}
  \]

  (Succeeds with the constraint \( c \geq 2 \).)
Randomized Quicksort

- Picks a pivot at random

- What is its complexity?
  - For randomized algorithms, we talk about *expected complexity*, which is an average over all possible values of the random variable.

- If pivot index is picked uniformly at random over the interval \([l, h]\), then:
  - every array element is equally likely to be selected as the pivot
  - every partition is equally likely
  - thus, *expected* complexity of *randomized* quicksort is given by the same recurrence as the *average* case of *qs*. 
Lower bounds for comparison-based sorting

- Sorting algorithms can be depicted as trees: each leaf identifies the input permutation that yields a sorted order.

![Sorting tree example](image)

- The tree has \( n! \) leaves, and hence a height of \( \log n! \). By Stirling’s approximation, \( n! \approx \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \), so, \( \log n! = O(n \log n) \)

- No **comparison-based** sorting algorithm can do better!
Bucket sort

**Overview**

**Divide:** Partition input into intervals (buckets), based on key values
- Linear scan of input, drop into appropriate bucket

**Recurse:** Sort each bucket

**Combine:** Concatenate bin contents

Images from Wikipedia commons
Bucket sort (Continued)

- Bucket sort generalizes quicksort to multiple partitions
  - Combination = concatenation
  - Worst case quadratic bound applies
  - But performance can be much better if input distribution is uniform.
    
    \textit{Exercise:} What is the runtime in this case?
  
- Used by letter sorting machines in post offices
Counting Sort

Special case of bucket sort where each bin corresponds to an interval of size 1.

- No need to recurse. Divide = conquered!
- Makes sense only if range of key values is small (usually constant)
- Thus, counting sort can be done in $O(n)$ time!

  *Hmm. How did we beat the $O(n\log n)$ lower bound?*
Radix Sorting

- Treat an integer as a sequence of digits
- Sort digits using counting sort
  - **LSD sorting:** Sort first on least significant digit, and most significant digit last. After each round of counting sort, results can be simply concatenated, and given as input to the next stage.
  - **MSD sorting:** Sort first on most significant digit, and least significant digit last. Unlike LSD sorting, we cannot concatenate after each stage.

**Note:** Radix sort does not divide inputs into smaller subsets. If you think of input as multi-dimensional data, then we break down the problem to each dimension.
Stable sorting algorithms

- **Stable sorting algorithms**: don’t change order of equal elements.

- Merge sort and LSD sort are stable. Quicksort is not stable.

Why is stability important?

- Effect of sorting on attribute $A$ and then $B$ is the same as sorting on $\langle B, A \rangle$.

- LSD sort won’t work without this property!

- Other examples: sorting spread sheets or tables on web pages.
Sorting strings

- Can use LSD or MSD sorting
  - Easy if all strings are of same length.
  - Requires a bit more care with variable-length strings.
    Starting point: use a special terminator character \( t < a \) for all valid characters \( a \).

- Easy to devise an \( O(nl) \) algorithm, where \( n \) is the number of strings and \( l \) is the maximum size of any string.
- But such an algorithm is not linear in input size.

**Exercise:** Devise a linear-time string algorithm.

Given a set \( S \) of strings, your algorithm should sort in \( O(|S|) \) time, where

\[
|S| = \sum_{s \in S} |s|
\]
Select $k^{th}$ largest element

Obvious approach: Sort, pick $k^{th}$ element — wasteful, $O(n \log n)$

Better approach: Recursive partitioning, search only on one side

```
qsel(A, l, h, k)
  if $l = h$ return $A[l];$
  $(h_1, l_2) =$
  partition(A, l, h);
  if $k \leq h_1$
    return $qsel(A, l, h_1, k)$
  else return $qsel(A, l_2, h, k)$
```

Complexity

Best case: Splits are even: $T(n) = n + T(n/2)$, which has an $O(n)$ solution.

Skewed 10%/90% $T(n) \leq n + T(0.9n)$ — still linear

Worst case: $T(n) = n + T(n-1)$ — quadratic!
Worst-case $O(n)$ Selection

**Intuition:** Spend a bit more time to select a pivot that ensures reasonably balanced partitions

**MoM Algorithm** [Blum, Floyd, Pratt, Rivest and Tarjan 1973]

---

**Abstract**

The number of comparisons required to select the i-th smallest of n numbers is shown to be at most a linear function of n by analysis of a new selection algorithm -- PICK. Specifically, no more than $5.4305^2$ n comparisons are ever required. This bound is improved for
**O(n)** Selection: MoM Algorithm

**Divide:**
- Divide into sets of 5 elements
- Compute median of each set (constant time)

**Recurse:** on the set of medians

**Combine:** No need – just a single subproblem

**Theorem:** MoM-based split won’t be worse than 30%/70%

**Result:** Guaranteed linear-time algorithm!

**Caveat:** The constant factor is non-negligible; use as fall-back if random selection repeatedly yields unbalanced splits.

**Note:** Till now, we looked at “embarrassingly divisible” problems. Starting with selection, we examine problems requiring increasing sophistication to set up the division.
Selecting minimum element: Priority Queues

Heap

- A tree-based data structure for priority queues

**Heap property:** $H$ is a heap if for every subtree $h$ of $H$

$$
\forall k \in \text{keys}(h) \quad \text{root}(h) \leq k
$$

where $\text{keys}(h)$ includes all keys appearing within $h$

**Note:** No ordering of siblings or cousins

- Supports $\text{insert}, \text{deleteMin}$ and $\text{min}$.
- Typically implemented using arrays, i.e., without an explicit tree data structure

Task of maintaining min is distributed to subsets of the entire set; alternatively, it can be thought of as maintaining several parallel queues with a single head.
Binary heap

Array representation: Store the heap elements in a breadth-first order in the array. Node $i$’s children will be at indices $2 \times i$ and $2 \times i + 1$

Min: Element at the root of the array, extracted in $O(1)$ time

DeleteMin: Overwrite root with last element of heap. Fix heap – takes $O(\log n)$ time, since only the ancestors of the last node need to be fixed up.

Insert: Append element to the end of array, fix up heap

MkHeap: Fix up the entire heap. Takes $O(n)$ time.

Heapsort: $O(n \log n)$ algorithm, MkHeap followed by $n$ calls to DeleteMin
Finding closest pair of points

*Problem:* Given a set of $n$ points in a $d$-dimensional space, identify the two that have the smallest Euclidean distance between them.

*Applications:* A central problem in graphics, vision, air-traffic control, navigation, molecular modeling, and so on.
Divide-and-conquer closest pair (2D)

**Divide:** Identify $k$ such that the line $x = k$ divides the points evenly. (Median computation, takes $O(n)$ time.)

**Recursive case:** Find closest pair in each half.

**Combine:**
- Can’t just take the min of the closest pairs from two halves.
- Need to consider pairs across the divide line — seems that this will take $O(n^2)$ time!
Observation (Key Observation 1)

Let $\delta_1$ and $\delta_2$ be the minimum distances in each half.

Need only consider points within $\delta = \min(\delta_1, \delta_2)$ from the dividing line.

We expect that only a small number of points will be within such a narrow strip.

But in the worst case, every point could be within the strip!
**Sparsity condition**

Consider a point \( p \) on the left \( \delta \)-strip. How many points \( q_1, ..., q_r \) on the right \( \delta \)-strip could be within \( \delta \) from \( p \)?

**Observation (Key Observation 2)**

- \( q_1, ..., q_r \) should all be within a rectangular \( 2\delta \times \delta \) as shown

- \( r \) can’t be too large: \( q_1, ..., q_r \) will crowd together, closer than \( \delta \)

**Theorem:** \( r \leq 6 \)

*We need to consider at most \( 6n \) cross-region pairs!*

Remains \( O(n) \) in higher dimensions as well
Closest pair: Summary

- **Recurrence:** \( T(n) = 2T(n/2) + \Omega(n) \), since median computation is already linear-time. Thus, \( T(n) = \Omega(n \log n) \).

- To get to \( O(n \log n) \), need to
  1. compute the \( \delta \)-strip in \( O(n) \) time
     - Keep the points in each region sorted in \( x \)-dimension
     - Takes an additional \( O(n \log n) \) time, no change to overall complexity
  2. compute \( q_1, \ldots, q_6 \) in \( O(1) \) time.
     - keep points in each region sorted *also* in \( y \)-dimension
     - maintain this order while deleting points outside \( \delta \) strip
     - in this list, for each \( p \), consider only 12 neighbors — 6 on each side of divide
The product $Z$ of two $n \times n$ matrices $X$ and $Y$ is given by

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

— leads to an $O(n^3)$ algorithm.

This follows by taking expected values of both sides of the following statement:

Time taken on an array of size $n \leq$ (time taken on an array of size $3n/4$) + (time to reduce array size to $\leq 3n/4$), and, for the right-hand side, using the familiar property that the expectation of the sum is the sum of the expectations.

From this recurrence we conclude that $T(n) = O(n)$: on any input, our algorithm returns the correct answer after a linear number of steps, on the average.

The Unix `sort` command

Comparing the algorithms for sorting and median-finding we notice that, beyond the common divide-and-conquer philosophy and structure, they are exact opposites. Mergesort splits the array in two in the most convenient way (first half, second half), without any regard to the magnitudes of the elements in each half; but then it works hard to put the sorted subarrays together. In contrast, the median algorithm is careful about its splitting (smaller numbers first, then the larger ones), but its work ends with the recursive call.

Quicksort is a sorting algorithm that splits the array in exactly the same way as the median algorithm; and once the subarrays are sorted, by two recursive calls, there is nothing more to do. Its worst-case performance is $\Theta(n^2)$, like that of median-finding. But it can be proved (Exercise 2.24) that its average case is $O(n \log n)$; furthermore, empirically it outperforms other sorting algorithms. This has made quicksort a favorite in many applications—for instance, it is the basis of the code by which really enormous files are sorted.
Divide-and-conquer Matrix Multiplication

Divide $X$ and $Y$ into four $n/2 \times n/2$ submatrices

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

Recursively invoke matrix multiplication on these submatrices:

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

Divided, but did not conquer! $T(n) = 8T(n/2) + O(n^2)$, which is still $O(n^3)$
Strassen’s Matrix Multiplication

Strassen showed that 7 multiplications are enough:

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

where

\[
P_1 = A(F - H) \\
P_2 = (A + B)H \\
P_3 = (C + D)E \\
P_4 = D(G - E) \\
P_5 = (A + D)(E + H) \\
P_6 = (B - D)(G + H) \\
P_7 = (A - C)(E + F)
\]

Now, the recurrence \( T(n) = 7T(n/2) + O(n^2) \) has \( O(n^{\log_2 7} = n^{2.81}) \) solution!

Best-to-date complexity is about \( O(n^{2.4}) \), but this algorithm is not very practical.
Karatsuba’s Algorithm

Same high-level strategy as Strassen — but predates Strassen.

Divide: $n$-digit numbers into halves, each with $n/2$-digits:

$$x = \begin{bmatrix} x_L & x_R \end{bmatrix} = 2^{n/2} x_L + x_R$$

$$y = \begin{bmatrix} y_L & y_R \end{bmatrix} = 2^{n/2} y_L + y_R$$

$$xy = 2^n x_L y_L + 2^{n/2}(x_L y_R + y_L x_R) + x_L y_L$$

Recursively compute $x_L y_L$, $x_R y_R$ and $(x_L + x_R)(y_L + y_R)$.

Combine: Key point — 3 calls to $n/2$-digit mult. is enough:

$$x_L y_R + y_L x_R = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$$

Now, recurrence $T(n) = 3T(n/2) + O(n)$ has an $O(n^{\log_2 3} = n^{1.59})$ solution!

Note: This trick for using 3 (not 4) multiplications noted by Gauss (1777-1855) in the context of complex numbers.
Fast Fourier Transformation

One of the most widely used algorithms — yet most people are unaware of its use!

**Solving differential equations:** Applied to many computational problems in engineering, e.g., heat transfer

**Audio:** MP3, digital audio processors, music/speech synthesizers, speech recognition, ...

**Image and video:** JPEG, MPEG, vision, ...

**Communication:** modulation, filtering, radars, software-defined radios, H.264, ...

**Medical diagnostics:** MRI, PET, ultrasound, ...

**Quantum computing:** See text Ch. 10

**Other:** Optics, data compression, seismology, ...
Theorem (Fourier Theorem)

Any (sufficiently smooth) function with a period $T$ can be expressed as a sum of series of sinusoids with periods $T/n$ for integral $n$.

$$a(t) = \sum_{n=0}^{\infty} \left( d_n \sin \left( \frac{2\pi nt}{T} \right) + e_n \cos \left( \frac{2\pi nt}{T} \right) \right)$$
Using the identity
\[ e^{ix} = \cos x + i \sin x \]
Fourier series becomes
\[ a(t) = \sum_{n=-\infty}^{\infty} c_n e^{2\pi i nt/T} \]
It can be shown
\[ c_n = \int_{0}^{T} a(t) e^{-2\pi int} dt \]
For real \( a(t) \), \( c_n = c_{-n}^* \).
Fourier Transform

- What if \( a \) is not periodic?

- May be we can start with the Fourier series definition for \( c_n \)

\[
    c_n = \int_0^T a(t)e^{-2\pi int} dt
\]

and let \( T \to \infty \)?

- Frequencies are not discrete any more, as the “fundamental frequency” \( f = 1/T \to 0 \)

- Instead of discrete coefficients \( c_n \), we will have a continuous function — call it \( s(f) \).

\[
    s(f) = \int_{-\infty}^{\infty} a(t)e^{-2\pi ift} dt
\]

- \( \mathcal{F}(a) \) denotes \( a \)’s Fourier transform

- \( \mathcal{F} \) is almost self-inverting:

\[
    \mathcal{F}(\mathcal{F}(a(t))) = a(-t)
\]
How do Fourier Series/Transform help?

**Differential equations**: Turn non-integrable functions into a sum of easily integrable ones.

**Some problems easier to solve in frequency domain**:

- **Filtering**: filter out noise, tuning, ...
- **Compression**: eliminate high frequency components, ...
- **Convolution**: Convolution in time domain becomes (simpler) multiplication in frequency domain.

**Definition (Convolution)**

\[(a \ast b)(t) = \int_{-\infty}^{\infty} a(t - x)b(x)dx\]

**Theorem (Convolution)**

\[\mathcal{F}(a \ast b)(t) = \mathcal{F}(a(t))\mathcal{F}(b(t))\]
Discrete Fourier Transform

- Real-world signals are typically sampled
  - DFT is a formulation of FT applicable to such samples

*Nyquist rate*: A signal with highest frequency $n/2$ can be losslessly reconstructed from $n$ samples.

- DFT of time domain samples $a_0, \ldots, a_{n-1}$ yields frequency domain samples $s_0, \ldots, s_{n-1}$:

$$s_f = \sum_{t=0}^{n-1} a_t e^{-2\pi ift/n} \quad \text{cf.} \quad s(f) = \int_{-\infty}^{\infty} a(t)e^{-2\pi ift} \, dt$$

**Note**: DFT formulation can be derived from FT by treating the sampling process as a multiplication by a sequence of impulse functions separated by the sampling interval
Background: Complex Plane, Polar Coordinates

The complex plane

\[ z = a + bi \text{ is plotted at position } (a, b). \]

Polar coordinates: rewrite as \( z = r(\cos \theta + i \sin \theta) = re^{i\theta} \), denoted \((r, \theta)\).

- **length** \( r = \sqrt{a^2 + b^2} \).
- **angle** \( \theta \in [0, 2\pi) \): \( \cos \theta = a/r, \sin \theta = b/r \).
- \( \theta \) can always be reduced modulo \( 2\pi \).

Examples:

<table>
<thead>
<tr>
<th>Number</th>
<th>Polar coords</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-1)</td>
<td>((1, \pi))</td>
</tr>
<tr>
<td>(i)</td>
<td>((1, \pi/2))</td>
</tr>
<tr>
<td>(5 + 5i)</td>
<td>((5\sqrt{2}, \pi/4))</td>
</tr>
</tbody>
</table>
Polar Coordinates and Multiplication

Multiply the lengths and add the angles:
\[(r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2).\]

For any \(z = (r, \theta),\)
- \(-z = (r, \theta + \pi)\) since \(-1 = (1, \pi).\)
- If \(z\) is on the unit circle (i.e., \(r = 1\)), then \(z^n = (1, n\theta).\)
Roots of unity on Complex Plane

Solutions to the equation $z^n = 1$.

By the multiplication rule: solutions are $z = (1, \theta)$, for $\theta$ a multiple of $2\pi/n$ (shown here for $n = 16$).

For even $n$:
- These numbers are plus-minus paired: $-(1, \theta) = (1, \theta + \pi)$
- Their squares are the $(n/2)$nd roots of unity, shown here with boxes around them.
Matrix representation of DFT

- Given time domain samples $a_t$ for $t = 0, 1, \ldots, n-1$,
- Compute frequency domain samples $s_f$ for $f = 0, 1, \ldots, n-1$

$$s_f = \sum_{t=0}^{n-1} a_t e^{-2\pi i f t / n} = \sum_{t=0}^{n-1} a_t \left(e^{-2\pi i / n}\right)^{ft} = \sum_{t=0}^{n-1} a_t \omega^{ft}$$

where $\omega = e^{-2\pi i / n}$ is the $n$th complex root of unity

$$\begin{bmatrix}
    s_0 \\
    s_1 \\
    s_2 \\
    \vdots \\
    s_j \\
    \vdots \\
    s_{n-1}
\end{bmatrix} =
\begin{bmatrix}
    1 & 1 & 1 & \ldots & 1 \\
    1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\
    1 & \omega^2 & \omega^4 & \ldots & \omega^{2(n-1)} \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    1 & \omega^j & \omega^{2j} & \ldots & \omega^{j(n-1)} \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)(n-1)}
\end{bmatrix}
\begin{bmatrix}
    a_0 \\
    a_1 \\
    a_2 \\
    \vdots \\
    a_j \\
    \vdots \\
    a_{n-1}
\end{bmatrix}$$
Matrix representation of DFT

Note that $e^{-2\pi i/n}$ is represents the $n^{\text{th}}$ root of 1, denoted $\omega$.

$$\begin{bmatrix}
    s_0 \\
    s_1 \\
    s_2 \\
    \vdots \\
    s_j \\
    \vdots \\
    s_{n-1}
\end{bmatrix}
= 
\begin{bmatrix}
    1 & 1 & 1 & \cdots & 1 \\
    1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\
    1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(n-1)} \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)}
\end{bmatrix}
\begin{bmatrix}
    a_0 \\
    a_1 \\
    a_2 \\
    \vdots \\
    a_j \\
    \vdots \\
    a_{n-1}
\end{bmatrix}$$

Possible interpretations of these matrix equations:

- Simultaneous equations that can be solved
- Change of basis (rotate coordinate system)
- Evaluation of polynomial $\sum_{k=0}^{n-1} a_k x^k$ at $x = \omega^j$, $0 \leq j < n$. 
Speeding up FFT Computation

- Matrix multiplication formulation has an obvious divide-and-conquer implementation

\[
M_n \overrightarrow{A}_{0 \ldots (n-1)} = \begin{bmatrix}
    M_{n/2}^{11} & M_{n/2}^{12} \\
    M_{n/2}^{21} & M_{n/2}^{22}
\end{bmatrix}
\begin{bmatrix}
    \overrightarrow{A}_{0 \ldots [n/2]} \\
    \overrightarrow{A}_{[n/2] \ldots (n-1)}
\end{bmatrix}
\]

But this algorithm still takes \(O(n^2)\) time

- \textit{... but wait!} — there are only \(O(n)\) distinct elements in the square matrix \(M_n\).

- \(O(n)\) repetitions of each element in \(M_n\), so there is significant scope for sharing operations on submatrices!
Observations about $M(\omega)$

- Two successive columns differ by a factor $\omega^j$ in the $j^{th}$ row.
- Rows that are $n/2$ rows apart differ by a factor of $\omega^{kn/2}$ in the $k^{th}$ column.
- Note that $\omega^{n/2} = -1$, so they differ by a factor of $-1$ on odd columns, and are identical on even columns.
The fast Fourier transform function FFT \((a, \omega)\) inputs an array \(a = (a_0, a_1, \ldots, a_{n-1})\), for \(n\) a power of 2. The problem of size \(n\) is reduced to two subproblems of size \(n/2\) (for clarity, one pair of outputs \((j, j+n/2)\) is singled out):

The orthogonality property can be summarized in the single equation

\[ M M^* = n I, \]

since \((M M^*)_{ij}\) is the inner product of \(a_j\) and \(a_k\) with their \(j\)th rows multiplied through by \(\omega_j\) and \(-\omega_j\), respectively. Therefore the final product is the vector

\[ a \omega^j, a \omega^j, \ldots, a \omega^{j+n-2}. \]

The divide-and-conquer step of the FFT can be drawn as a very simple circuit. Here is how it works:

Only two subproblems of size \(n/2\):

Multiply \(M_{n/2}\) by \(\hat{A}_{odd}\) and \(\hat{A}_{even}\):

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

with an \(O(n \log n)\) solution.

But wait! \(M_n\) has \(O(n^2)\) size, how can we operate on it in \(O(n)\) time?

DFT Matrix Multiplication, Rearranged ...

\[ M_n(\omega) \]

\[ j \]

\[ \omega^{jk} \]

\[ a \]

\[ a_0 \]

\[ a_1 \]

\[ a_2 \]

\[ a_3 \]

\[ a_4 \]

\[ \vdots \]

\[ a_{n-1} \]

\[ j \]

\[ \omega^{jk} \]

\[ \omega^j \cdot \omega^{jk} \]

\[ a_0 \]

\[ a_2 \]

\[ \vdots \]

\[ a_{n-2} \]

\[ a_1 \]

\[ a_3 \]

\[ \vdots \]

\[ a_{n-1} \]

\[ j+n/2 \]

\[ \omega^{2jk} \]

\[ \omega^j \cdot \omega^{2jk} \]

\[ j+n/2 \]

\[ \omega^{2jk} \]

\[ -\omega^j \cdot \omega^{2jk} \]
**FFT Algorithm**

\[
\text{function } \text{FFT}(a, \omega) \\
\text{Input: } \text{An array } a = (a_0, a_1, \ldots, a_{n-1}), \text{ for } n \text{ a power of 2} \\
\text{A primitive } n\text{th root of unity, } \omega \\
\text{Output: } M_n(\omega) a
\]

if \( \omega = 1 \):  return \( a \)

\[
(s_0, s_1, \ldots, s_{n/2-1}) = \text{FFT}((a_0, a_2, \ldots, a_{n-2}), \omega^2) \\
(s', s'_1, \ldots, s'_{n/2-1}) = \text{FFT}((a_1, a_3, \ldots, a_{n-1}), \omega^2)
\]

for \( j = 0 \) to \( n/2 - 1 \):

\[
\begin{align*}
  r_j &= s_j + \omega^j s'_j \\
  r_{j+n/2} &= s_j - \omega^j s'_j
\end{align*}
\]

return \((r_0, r_1, \ldots, r_{n-1})\)
Convolution in the Discrete World

\[(\overrightarrow{A}_n \ast \overrightarrow{B}_m)_t = \sum_{x=0}^{m-1} a_{t-x} b_x\]

cf. \((a \ast b)(t) = \int_{-\infty}^{\infty} a(t-x)b(x)dx\)

Linear convolution: \(a_{t-x} = 0\) if \(x > t\)

Circular convolution: \(a_{t-x} = a_{t-x+n}\) if \(x > n\). (Equivalent to treating \(A\) as a periodic function.)

Zero-extended convolution: First extend \(A\) and \(B\) to have \(m + n - 1\) samples by letting \(A_y = 0\) for \(m \leq y < m + n\) and \(B_z = 0\) for \(n \leq z < m + n\).

With zero-extension, the definitions of linear and circular conventions match, and hence become equivalent. Hence, we will deal only with zero-extended convolution.

Theorem (Discrete Convolution) \(\mathcal{F}(\overrightarrow{A}_n \ast \overrightarrow{B}_m) = \mathcal{F}(\overrightarrow{A}_n)\mathcal{F}(\overrightarrow{B}_m)\)
Why this fascination with convolution?

- Computationally, convolution is a loop to add products
- The convolution theorem says we can replace this $O(n)$ loop by a single operation on the DFT. *That is fascinating!*
  - *Wait a minute!* What about the cost of computing $\mathcal{F}$ first?
- If we use FFT, then we the computation of $\mathcal{F}$ and its inversion will still obe $O(n \log n)$, not quadratic.

Can we use FFT as a building block to speed up algorithms for other problems?
- Integer multiplication looks like a convolution, and usually takes $O(n^2)$. Can we make it $O(n \log n)$?
Integer Multiplication Revisited

- An integer represented using digits
  \[ b_{n-1} \ldots b_0 \]
  where \( 0 \leq b_i < d \) is very similar to the polynomial
  \[
  B(x) = \sum_{i=0}^{n-1} b_i x^i
  \]
  Specifically, the value of the integer is \( B(d) \).

- Integer multiplication follows the same steps as polynomial multiplication:
  \[
  b_{n-1} \ldots b_0 \times b'_{n-1} \ldots b'_0 = (B(x) \times B'(x))(d)
  \]
Polynomials: Basic Properties

Horner's rule

An \( n \)th degree polynomial \( \sum_{i=0}^{n} a_i x^i \) can be evaluated in \( O(n) \) time:

\[
((\cdots ((a_n x + a_{n-1}) x + a_{n-2}) x + \cdots + a_1) x + a_0)
\]

Roots and Interpolation

- An \( n \)th degree polynomial \( A(x) \) has exactly \( n \) roots \( r_1, \ldots, r_n \). In general, \( r_i \)'s are complex and need not be distinct.
- It can be represented as a product of sums using these roots:

\[
A(x) = \sum_{i=1}^{n} a_i x^i = \prod_{i=0}^{n} (x - r_i)
\]

- Alternatively, \( A(x) \) can be specified uniquely by specifying \( n + 1 \) points \((x_i, y_i)\) on it, i.e., \( A(x_i) = y_i \).
Operations on Polynomials

<table>
<thead>
<tr>
<th>Representation</th>
<th>Add</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coefficients</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Roots</td>
<td>?</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Points</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Note: Point representation is the best for computation! But usually, only the coefficients are given.

Solution: Convert to point form by evaluating $A(x)$ at selected points.

But the conversion defeats the purpose! Conversion requires $O(n)$ evaluations, each taking $O(n)$ time, thus we are back to $O(n^2)$ total time.
FFT to the Rescue!

Matrix form of DFT and interpretation as polynomial evaluation:

\[
\begin{bmatrix}
    s_0 \\
    s_1 \\
    \vdots \\
    s_j \\
    \vdots \\
    s_{n-1}
\end{bmatrix}
= \begin{bmatrix}
    1 & 1 & 1 & \cdots & 1 \\
    1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(n-1)} \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)}
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
\vdots \\
a_j \\
\vdots \\
a_{n-1}
\end{bmatrix}
\]

- **Voila!** FFT computes \( A(x) \) at \( n \) points \( (x_i = \omega^i) \) in \( O(n \log n) \) time!

- **\( O(n \log n) \)** integer multiplication
  
  Convert to point representation using FFT \( O(n \log n) \)
  
  Multiply on point representation \( O(n) \)
  
  Convert back to coefficients using FFT\(^{-1} \) \( O(n \log n) \)
FFT to the Rescue!

$$\begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_j \\ \vdots \\ s_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_j \\ \vdots \\ a_{n-1} \end{bmatrix}$$

FFT can be thought of as a clever way to choose points:

- Evaluations at many distinct points “collapse” together

- This is why we are left with \(2T(n/2)\) work after division, instead of \(4T(n/2)\) for a naive choice of points.
**FFT-based Multiplication: Summary**

- FFT works with $2^k$ points — Increases work by up to 2x.
- Product of two $n^{th}$-degree polynomial has degree $2n$
  - We need to work with $2n$ points, i.e., 4x increase in time.

- Requires inverting to coefficient representation after multiplication:
  \[
  \vec{S}_n = M_n(\omega)\vec{A}_n
  \]
  \[
  M_n^{-1}(\omega)\vec{S}_n = M_n^{-1}(\omega)M_n(\omega)\vec{A}_n = \vec{A}_n
  \]
  It is easy to show that $M_n^{-1}(\omega) = M_n(-\omega)/n$, and hence:
  \[
  \vec{A}_n \ast \vec{B}_n = \text{FFT}(\text{FFT}(\vec{A}_{2n}, \omega) \cdot \text{FFT}(\vec{B}_{2n}, \omega), \omega^{-1})/n
  \]

*We are back to the convolution theorem!*
More careful analysis ...

- Computations on complex or real numbers can lose precision.
- For integer operations, we should work in some other ring — usually, we choose a ring based on modulo arithmetic.
- Ex: in mod 33 arithmetic, 2 is the 10th root of 1, i.e., $2^{10} \equiv 1 \pmod{33}$

More generally, 2 is the $n$th root of unity modulo $(2^{n/2} + 1)$

- Point-wise additions and multiplications are not $O(1)$.
  - We are adding up to $n$ numbers (“digits”) — the sum requires $\Omega(\log n)$ bits
  - So, total cost increases by at least $\log n$, i.e., $O(n \log^2 n)$.

[Schonhage-Strassen ’71] developed $O(n \log n \log \log n)$ algorithm: recursively apply their technique for “inner” operations.
### Integer Multiplication Summary

- **Algorithms implemented in libraries for arbitrary precision arithmetic, with applications in public key cryptography, computer algebra systems, etc.**

- GNU MP is a popular library, uses various algorithms based on input size: naive, Karatsuba, Toom-3, Toom-4, or Schonhage-Strassen (at about 50K digits).

- Karatsuba is Toom-2. Toom-N is based on
  - Evaluating a polynomial at \(2^N\) points,
  - performing point-wise multiplication, and
  - interpolating to get back the polynomial, while
  - minimizing the operations needed for interpolation