SPOTLIGHT ON

Complexity The achievement of Cooley and Tukey to reduce the complexity of the DFT from $O(n^2)$ operations to $O(n \log n)$ operations opened up a world of possibilities for Fourier transform methods. A method that scales "almost linearly" with the size of the problem is very valuable. For example, there is a possibility of using it for real-time data, since analysis can occur approximately at the same timescale that data are acquired. The development of the FFT was followed a short time later with specialized circuitry for implementing it, now represented by DSP chips for digital signal processing that are ubiquitous in electronic systems for analysis and control.

We will show how to compute $z = M_n x$ recursively. To complete the DFT requires dividing by $\sqrt{n}$, or $y = F_n x = z/\sqrt{n}$.

We start by showing how the $n = 4$ case works, to get the main idea across. The general case will then be clear. Let $\omega = e^{-i2\pi/4} = -i$. The Discrete Fourier Transform is

$$\begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}. \tag{10.14}$$

Write out the matrix product, but rearrange the order of the terms so that the even-numbered terms come first:

$$\begin{aligned} z_0 &= \omega^0 x_0 + \omega^0 x_2 + \omega^0(\omega^0 x_1 + \omega^0 x_3) \\ z_1 &= \omega^0 x_0 + \omega^2 x_2 + \omega^1(\omega^0 x_1 + \omega^2 x_3) \\ z_2 &= \omega^0 x_0 + \omega^4 x_2 + \omega^2(\omega^0 x_1 + \omega^4 x_3) \\ z_3 &= \omega^0 x_0 + \omega^6 x_2 + \omega^3(\omega^0 x_1 + \omega^6 x_3) \end{aligned}$$

Using the fact that $\omega^4 = 1$, we can rewrite these equations as

$$\begin{aligned} z_0 &= (\omega^0 x_0 + \omega^0 x_2) + \omega^0(\omega^0 x_1 + \omega^0 x_3) \\ z_1 &= (\omega^0 x_0 + \omega^2 x_2) + \omega^1(\omega^0 x_1 + \omega^2 x_3) \\ z_2 &= (\omega^0 x_0 + \omega^0 x_2) + \omega^2(\omega^0 x_1 + \omega^0 x_3) \\ z_3 &= (\omega^0 x_0 + \omega^2 x_2) + \omega^3(\omega^0 x_1 + \omega^2 x_3) \end{aligned}$$

Notice that each term in parentheses in the top two lines is repeated verbatim in the bottom two lines. Define

$$\begin{aligned} u_0 &= \mu^0 x_0 + \mu^0 x_2 \\ u_1 &= \mu^0 x_0 + \mu^1 x_2 \end{aligned}$$

and

$$\begin{aligned} v_0 &= \mu^0 x_1 + \mu^0 x_3 \\ v_1 &= \mu^0 x_1 + \mu^1 x_3, \end{aligned}$$

where $\mu = \omega^2$ is a 2nd root of unity. Both $u = (u_0, u_1)^T$ and $v = (v_0, v_1)^T$ are essentially DFTs with $n = 2$; more precisely,

$$\begin{aligned} u &= M_2 \begin{bmatrix} x_0 \\ x_2 \end{bmatrix} \\ v &= M_2 \begin{bmatrix} x_1 \\ x_3 \end{bmatrix}. \end{aligned}$$

We can write the original M_4x as

$$\begin{aligned} z_0 &= u_0 + \omega^0 v_0 \\ z_1 &= u_1 + \omega^1 v_1 \\ z_2 &= u_0 + \omega^2 v_0 \\ z_3 &= u_1 + \omega^3 v_1. \end{aligned}$$

In summary, the calculation of the DFT(4) has been reduced to a pair of DFT(2)s plus some extra multiplications and additions.

Ignoring the $1/\sqrt{n}$ for a moment, DFT(n) can be reduced to computing two DFT($n/2$)s plus $2n - 1$ extra operations ($n - 1$ multiplications and n additions). A careful count of the additions and multiplications necessary yields Theorem 10.5.

THEOREM 10.5 **Operation Count for FFT.** Let n be a power of 2. Then the Fast Fourier Transform of size n can be completed in $n(2\log_2 n - 1) + 1$ additions and multiplications, plus a division by $\sqrt{n}$. ■

Proof. Ignore the square root, which is applied at the end. The result is equivalent to saying that the DFT(2^m) can be completed in $2^m(2m - 1) + 1$ additions and multiplications. In fact, we saw above how a DFT(n), where n is even, can be reduced to a pair of DFT($n/2$)s. If n is a power of two—say, $n = 2^m$—then we can recursively break down the problem until we get to DFT(1), which is multiplication by the 1×1 identity matrix, taking zero operations. Starting from the bottom up, DFT(1) takes no operations, and DFT(2) requires two additions and a multiplication: $y_0 = u_0 + 1v_0$, $y_1 = u_0 + \omega v_0$, where u_0 and v_0 are DFT(1)s (that is, $u_0 = y_0$ and $v_0 = y_1$).

DFT(4) requires two DFT(2)s plus $2 * 4 - 1 = 7$ further operations, for a total of $2(3) + 7 = 2^m(2m - 1) + 1$ operations, where $m = 2$. We proceed by induction: Assume that this formula is correct for a given m. Then DFT(2^{m+1}) takes two DFT(2^m)s, which take $2(2^m(2m - 1) + 1)$ operations, plus $2 \cdot 2^{m+1} - 1$ extras (to complete equations similar to (10.15)), for a total of

$$\begin{aligned} 2(2^m(2m - 1) + 1) + 2^{m+2} - 1 &= 2^{m+1}(2m - 1 + 2) + 2 - 1 \\ &= 2^{m+1}(2(m + 1) - 1) + 1. \end{aligned}$$

Therefore, the formula $2^m(2m - 1) + 1$ operations is proved for the fast version of DFT(2^m), from which the result follows. ❐

The fast algorithm for the DFT can be exploited to make a fast algorithm for the inverse DFT without further work. The inverse DFT is the complex conjugate matrix $\overline{F}_n$. To carry out the inverse DFT of a complex vector y, just conjugate, apply the FFT, and conjugate the result, because

$$F_n^{-1} y = \overline{F}_n y = \overline{F_n \overline{y}}. \tag{10.15}$$

▸ ADDITIONAL EXAMPLES

1. (a) Compute the Discrete Fourier Transform of the vector $x = [1\ 2\ 3\ 2]$, and (b) apply the Inverse Discrete Fourier Transform to the result.

*2. (a) Compute the Fast Fourier Transform of $x = [5\ 10\ 5\ 0]$, and (b) apply the Inverse Fast Fourier Transform to the result.

Solutions for Additional Examples can be found at goo.gl/UGSy5G
(* example with video solution)

10.1 Exercises

Solutions for Exercises numbered in **blue** can be found at goo.gl/sXeGT5

1. Find the DFT of the following vectors: (a) $[0, 1, 0, -1]$ (b) $[1, 1, 1, 1]$ (c) $[0, -1, 0, 1]$ (d) $[0, 1, 0, -1, 0, 1, 0, -1]$
2. Find the DFT of the following vectors: (a) $[3/4, 1/4, -1/4, 1/4]$ (b) $[9/4, 1/4, -3/4, 1/4]$ (c) $[1, 0, -1/2, 0]$ (d) $[1, 0, -1/2, 0, 1, 0, -1/2, 0]$
3. Find the inverse DFT of the following vectors: (a) $[1, 0, 0, 0]$ (b) $[1, 1, -1, 1]$ (c) $[1, -i, 1, i]$ (d) $[1, 0, 0, 0, 3, 0, 0, 0]$
4. Find the inverse DFT of the following vectors: (a) $[0, -i, 0, i]$ (b) $[2, 0, 0, 0]$ (c) $[1/2, 1/2, 0, 1/2]$ (d) $[1, 3/2, 1/2, 3/2]$
5. (a) Write down all fourth roots of unity and all primitive fourth roots of unity. (b) Write down all primitive seventh roots of unity. (c) How many primitive pth roots of unity exist for a prime number p?
6. Prove Lemma 10.1.
7. Find the real numbers $a_0, a_1, b_1, a_2, b_2, \ldots, a_{n/2}$ as in (10.13) for the Fourier transforms in Exercise 1.
8. Prove that the matrix in (10.10) is the inverse of the Fourier matrix F_n.

10.2 TRIGONOMETRIC INTERPOLATION

What does the Discrete Fourier transform actually do? In this section, we present an interpretation of the output vector y of the Fourier transform as interpolating coefficients for evenly spaced data in order to make its workings more understandable.

10.2.1 The DFT Interpolation Theorem

Let $[c, d]$ be an interval and let n be a positive integer. Define $\Delta t = (d - c)/n$ and $t_j = c + j\Delta t$ for $j = 0, \ldots, n - 1$ to be evenly spaced points in the interval. For a given input vector x to the Fourier transform, we will interpret the component x_j as the jth component of a measured signal. For example, we could think of the components of x as a series of measurements, measured at the discrete, evenly spaced times t_j, as shown in Figure 10.4.

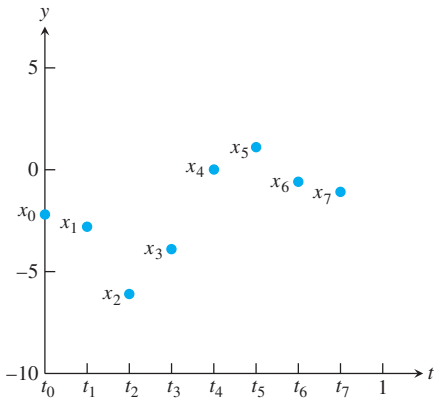


Figure 10.4 The components of x viewed as a time series. The Fourier transform is a way to compute the trigonometric polynomial that interpolates this data.

4 Randomized Analysis, Quicksort: Chapters 5 & 7

Reading: Sections 5.1–5.3 (probability, review), Chapter 7

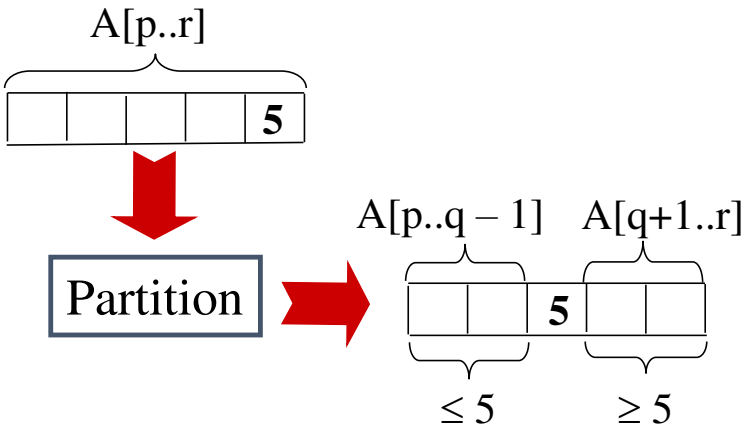
Quicksort: review

```
Quicksort(A, p, r)
    if p < r then
        q = Partition(A, p, r)
        Quicksort(A, p, q – 1)
        Quicksort(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

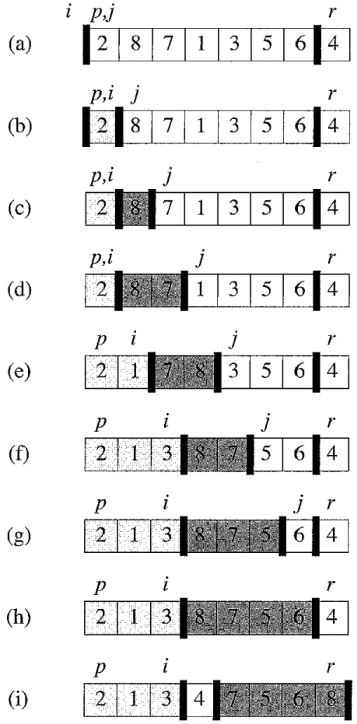
Quicksort: review

```
Quicksort(A, p, r)
    if p < r then
        q = Partition(A, p, r)
        Quicksort(A, p, q – 1)
        Quicksort(A, q + 1, r)
```



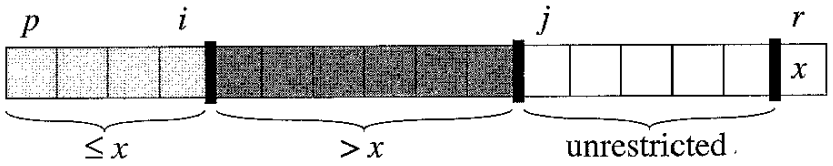

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

QuickSort



PARTITION(A, p, r)

```
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```



QuickSort

```
3   for j = p to r − 1
4       if A[j] ≤ x
5           i = i + 1
6           exchange A[i] with A[j]
```

Proof of Correctness: PARTITION

Loop invariant

At the beginning of each iteration of the loop (3-6), for any array index k:

1.If $p \le k \le i$, then $A[k] \le x$;

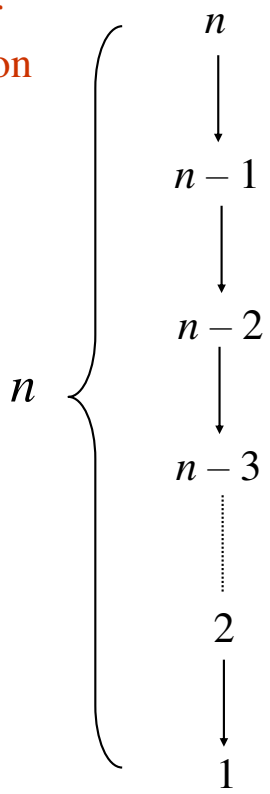
2.If $i+1 \le k \le j-1$, then $A[k] > x$;

3.If $k = r$, then $A[k] = x$.

4.If $j \le k \le r-1$, then we don't know anything about $A[k]$.

Worst-case Partition Analysis

Recursion tree for worst-case partition



Split off a single element at each level:

$T(n) = T(n-1) + T(0) + \text{PartitionTime}(n)$

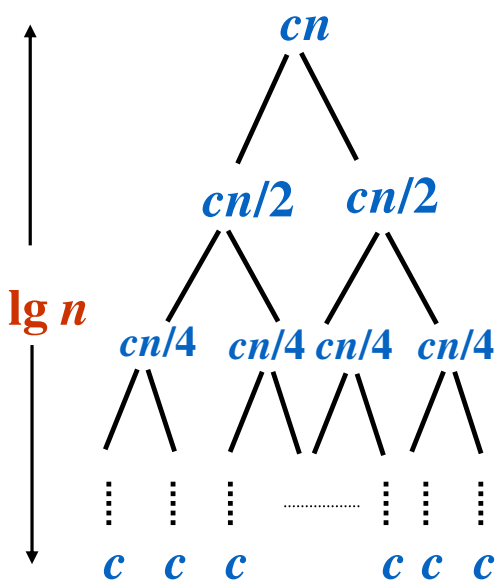
$= T(n-1) + \Theta(n)$

$= \sum_{k=1 \text{ to } n} \Theta(k)$

$= \Theta(\sum_{k=1 \text{ to } n} k)$

$= \Theta(n^2)$

Best-case Partitioning



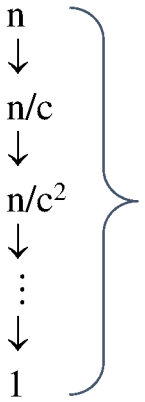
- Each subproblem size $\leq n/2$.
- Recurrence for running time
  - $T(n) \leq 2T(n/2) + \text{PartitionTime}(n)$
    $= 2T(n/2) + \Theta(n)$
- **$T(n) = \Theta(n \lg n)$**

Unbalanced Partition Analysis

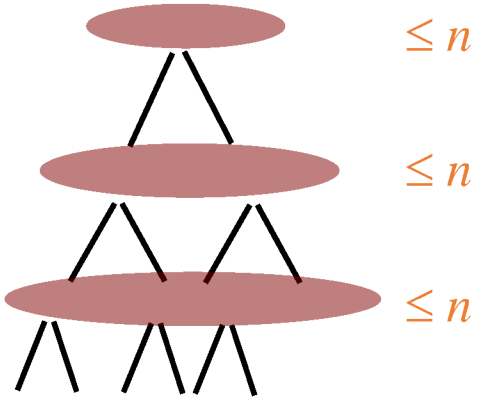
What happens if we get poorly-balanced partitions,
e.g., something like: $T(n) \le T(9n/10) + T(n/10) + \Theta(n)$?

Still $\Theta(n \lg n)$ as long as the split is constant fraction of n… HW 2, #2

Intuition: Can divide n by $c > 1$ only $\Theta(\lg n)$ times before getting 1.



Roughly $\log_c n$ levels;
Cost per level is $O(n)$.

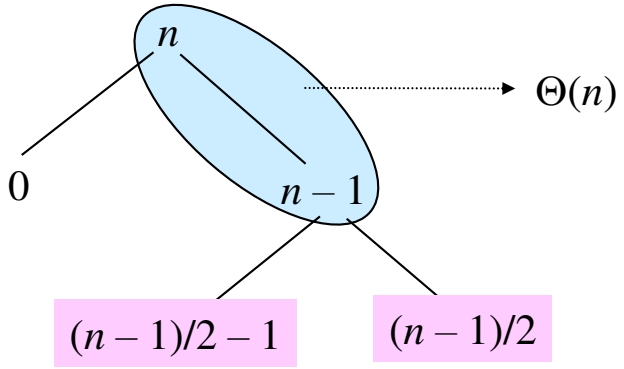


(**Remember:** Different base logs are related by a constant.)

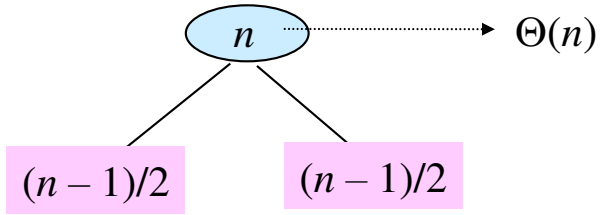
Intuition for the Average Case

- Partitioning is unlikely to happen in the same way at every level.
  - Split ratio is different for different levels.
    (Contrary to our assumption in the previous slide.)
- Partition produces a mix of "good" and "bad" splits, distributed randomly in the recursion tree.
- What is the running time likely to be in such a case?

Intuition for the Average Case

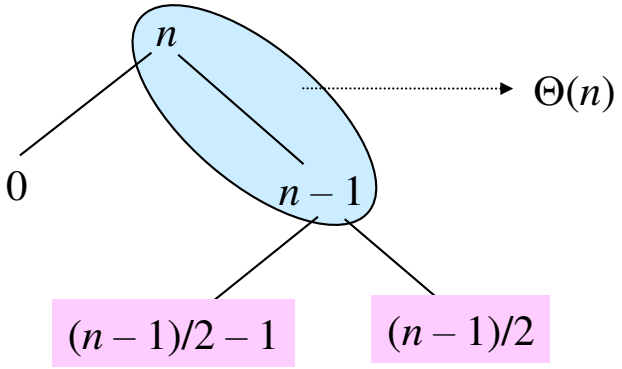


Bad split followed by a good split:
Produces subarrays of sizes 0, $(n-1)/2-1$, and $(n-1)/2$.
Cost of partitioning :
$\Theta(n) + \Theta(n-1) = \Theta(n)$.

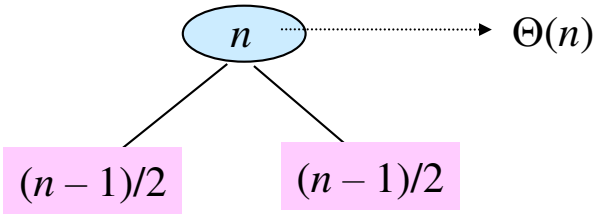


Good split at the first level:
Produces two subarrays of size $(n-1)/2$.
Cost of partitioning :
$\Theta(n)$.

What is the running time?



Bad split followed by a good split:
Produces subarrays of sizes 0,
$(n-1)/2-1$, and $(n-1)/2$.
Cost of partitioning :
$\Theta(n) + \Theta(n\text{-}1) = \Theta(n)$.



Good split at the first level:
Produces two subarrays of size $(n-1)/2$.
Cost of partitioning :
$\Theta(n)$.

Situation at the end of case 1 is not worse than that at the end of case 2.
When splits alternate between good and bad,
cost of bad split can be absorbed into the cost of good split.
Running time is $O(n \lg n)$, *with larger hidden constants*

Randomized Quicksort

- Want to make running time independent of input ordering.
- How can we do that?

Randomized Quicksort

- Want to make running time independent of input ordering.
- How can we do that?
  - » Make the algorithm randomized.

Randomized Quicksort

- Want to make running time independent of input ordering.
- How can we do that?
  - » Make the algorithm randomized.
  - » Make every possible input equally likely.

Randomized Quicksort

- Want to make running time independent of input ordering.
- How can we do that?
  - » Make the algorithm randomized.
  - » Make every possible input equally likely.
    - Can randomly shuffle to permute the entire array.
    - For quicksort, it is sufficient if we can ensure that every element is equally likely to be the *pivot*.
    - So, we choose an element in $A[p..r]$ and exchange it with $A[r]$.
    - Because the *pivot* is randomly chosen, we expect the partitioning to be well balanced on average.

Randomized Version

Want to make running time independent of input ordering.

```
Randomized-Partition(A, p, r)
    k:= Random(p, r);
    A[r] ↔ A[k];
    Partition(A, p, r)
```

```
Randomized-Quicksort(A, p, r)
    if p < r then
        q := Randomized-Partition(A, p, r);
        Randomized-Quicksort(A, p, q – 1);
        Randomized-Quicksort(A, q + 1, r)
    fi
```