CSC373: Algorithm Design, Analysis and Complexity Winter/Spring 2020

Allan Borodin and Sara Rahmati

January 6-10, 2020

Introduction

Course Organization: See General Course Info on course web site:
`http://www.cs.toronto.edu/∼bor/373s20/`

Note: While we may post lecture slides, we will not be relying on lecture slides but mainly relying on the white/black board and the texts. There are (at least) three excellent texts for material in this course. We are using CLRS, DPV, and KT. There may also be some additional material that we will post. We are hoping for aninteractive class with everyone reading the suggested sections of these three texts and any recommended additional material.

TODO: We need to assign students to tutorials. Once we have been assigned rooms for the tutorials, we will assign students to tutorials by birthdays. We will (depending on having enough TAas) initially start with 2 or 3 tutorial sections (4-5 Monday for the day time class and 5-7 for the evening class). The first tutorials will take place on Monday, January 13. Please check the web page for your assigned tutorial room if this has not been announced in class.

What is CSC373?

- CSC373 is a "compromise course". Namely, in the desire to give students more choice, there are only two specific courses which are required for all CS specialists. Namely we require one "systems course" CSC369 and one "theory course" CSC373 whereas in the past we required both an algorithms course and a complexity course. Our solution was to make CSC373 mainly an algorithms course, but to also include an introduction to complexity theory. DCS also provides a 4th year complexity course CSC465 as well as a more advanced undergraduate algorithms course CSC473.
- The complexity part of the course relies on suitable "reductions" and "transfomations" (i.e., converting an instance of a problem A to an instance of problem B). As such, *since reductions and transformations are algorithms*, this is not an unnatural combination. The main difference is that we generally use reductions in complexity theory to provide evidence that something is difficult (rather than use it to derive new algorithms). More on this later. Indeed most algorithm textbooks include NP-completeness.

The dividing line between efficiently computable and NP hardness

- Many closely related problems are such that:

One problem has an efficient algorithm (e.g., polynomial time) while a variant becomes (according to “well accepted” conjectures) difficult to compute (e.g. requiring exponential time complexity).

- For example:
  - Interval Scheduling vs Job Interval Scheduling
  - Minimum Spanning Tree (MST) vs Bounded degree MST
  - MST vs Steiner tree
  - Shortest paths vs Longest (simple) paths
  - 2-Colourability vs 3-Colourability

- Our focus is worst case analysis in contrast to peformance “in practice”. Why is “practice” in quotes?

Comments and disclaimers on the course perspective; what this course is and is not about

- As this is a required basic course, we are following the perspective of the standard CS undergraduate texts. However, we may sometimes introduce ideas relating to current research.
- Most CS undergraduate algorithm courses/texts study the high level presentation of algorithms within the framework of *basic algorithmic paradigms* that can be applied in a wide variety of contexts.
- Moreover, the perspective is one of studying “worst case” (adversarial) input instances.

Comments and disclaimers on the course perspective; what this course is and is not about

- As this is a required basic course, we are following the perspective of the standard CS undergraduate texts. However, we may sometimes introduce ideas relating to current research.
- Most CS undergraduate algorithm courses/texts study the high level presentation of algorithms within the framework of *basic algorithmic paradigms* that can be applied in a wide variety of contexts.
- Moreover, the perspective is one of studying "worst case" (adversarial) input instances.
  Why not study "average case analysis"

Comments and disclaimers on the course perspective; what this course is and is not about

- As this is a required basic course, we are following the perspective of the standard CS undergraduate texts. However, we may sometimes introduce ideas relating to current research.
- Most CS undergraduate algorithm courses/texts study the high level presentation of algorithms within the framework of *basic algorithmic paradigms* that can be applied in a wide variety of contexts.
- Moreover, the perspective is one of studying "worst case" (adversarial) input instances.
  Why not study "average case analysis"
- A more applied perspective (i.e., an "algorithmic engineering" course that say discusses implementations of algorithms in industrial applications) is beyond the scope of this course.

Comments and disclaimers on the course perspective; what this course is and is not about

- As this is a required basic course, we are following the perspective of the standard CS undergraduate texts. However, we may sometimes introduce ideas relating to current research.
- Most CS undergraduate algorithm courses/texts study the high level presentation of algorithms within the framework of *basic algorithmic paradigms* that can be applied in a wide variety of contexts.
- Moreover, the perspective is one of studying "worst case" (adversarial) input instances.
  Why not study "average case analysis"
- A more applied perspective (i.e., an "algorithmic engineering" course that say discusses implementations of algorithms in industrial applications) is beyond the scope of this course.
  Why isn't such a course offered?

What this course is and is not about (continued)

- Our focus is on deterministic algorithms for discrete combinatorial (and some numeric/algebraic) type problems which we study with respect to *sequential time* within a von Neumann RAM computational model.
- Even within theoretical CS, there are many focused courses and texts for particular subfields. At an advanced undergraduate or graduate level, we might have entire courses on for example, randomized algorithms, stochastic (i.e., "average case") analysis, approximation algorithms, linear programming (and more generally mathematical programming), online algorithms, parallel algorithms, streaming algorithms, sublinear time algorithms, spectral algorithms (and more, generally algebraic algorithms), geometric algorithms, continuous methods for discrete problems, genetic algorithms, etc.

The growing importance of TCS

- Core questions (e.g. P vs NP) have gained prominence in both the intellectual and popular arenas.
- There are relatively recent breakthroughs in faster algorithms and scalable parallelizable data structures and algorithms, complexity based cryptography, approximate combinatorial optimization, pseudo-randomness, coding theory,. . .
- TCS has expanded its frontiers.
  Many fields rely increasingly on the algorithms and abstractions of TCS, creating new areas of inquiry within theory and new fields at the boundaries between TCS and disciplines such as:
  - computational biology
  - algorithmic game theory
  - algorithmic aspects of social networks
  - social choice theory

End of introductory comments

We recognize some (many, most?) students may be attending only because it is required. You may also be wondering "will I ever use any of the material in this course"? or "Why is this course required"?

End of introductory comments

We recognize some (many, most?) students may be attending only because it is required. You may also be wondering "will I ever use any of the material in this course"? or "Why is this course required"?

How many share this sentiment?

End of introductory comments

We recognize some (many, most?) students may be attending only because it is required. You may also be wondering “will I ever use any of the material in this course”? or “Why is this course required”?

How many share this sentiment?

Our goal is to instill some more analytical, precise ways of thinking and this goes beyond the specific course content. The Design and Analysis of Algorithms is a required course is almost all North American CS programs. (It probably is also required throughout the world but we know more about North America.) So the belief that this kind of thinking is useful and important is widely accepted. We hope that we can make it seem meaningful to you now and not just maybe only 10 years from now.

Tentative set of topics)

- Introduction and Motivation (We just did it.)
- Divide and Conquer (1 week)
- Greedy algorithms (1-2 weeks)
- Dynamic Programming (1-2 weeks)
- Network flows; bipartite matching (1-2 weeks)
- NP and NP-completeness; self reduction (2-3 weeks)
- Approximation algorithms (to be mentioned throughout term)
- Linear Programming; IP/LP rounding (2 weeks)
- Local search (1 week)
- Randomized algorithms (1 week)

Outline for the course content in Week 1

We will start the course with divide and conquer, a basic algoithmic paradigm that you are familiar with from say CSC236/CSC240, and CSC263/CSC265.
The texts contain many examples of divide and conquer as well as how to solve certain types of recurrences which again you have already seen in previous courses. So we do not plan to spend too much time on divide and conquer.
Here is what we will be doing:
(1) An informal statement of the divide and conquer paradigm
Note: Like other paradigms we will consider, we do not present a precise definition for divide and conquer. For our purposes (and that of the texts), it is a matter of "you know it when you see it". But if we wanted to say prove that a given problem could not be solved efficiently by (for example) a divide and conquer algorithm we would need a precise model.

Outline for Week 1 continued

(2) We will choose a few of the many examples taken mainly from the examples in texts:

- CLRS: maximum subarray, Strassen's matrix multiplication, quicksort, median and selection in sorted list, dynamic multithreading, the FFT algorithm, closest pair in $\mathbb{R}^2$, sparse cuts in graphs
- KT: merge sort, counting imversions, closest pair in $\mathbb{R}^2$, integer multiplication, FFT, quicksort, medians and selection

(3) The typical recurrences; the "master theorem".

(4) Comments on choosing the right abstraction of the problem.

The divide and conquer paradigm

As roughly stated in DPV chapter 2, the divide and conquer paradigm solves a problem by:

1. Dividing the problem into smaller subproblems of the same type
   Note: In some cases we have to generalize the given problem so as to lend itself to the paradigm. We will also see this need to generalize in the dynamic programming paradigm.
   Aside: The need to generalize is one reason why it is hard to formnalize these paradigms.
2. Recursively solving these subproblems
3. Combining the results from the subproblems

Counting inversions from Kevin Wayne's slides.

Kevin Wayne (at Princeton University) has excellent slides for the material in the Kleinberg and Tardos text. See
`http://www.cs.princeton.edu/~wayne/kleinberg-tardos`

As a first example of divide and conquer, we will start with the problem of counting inversions in an unsorted array. That is, for how many pairs (i, j) is $a_i > a_j$? (As mentioned in Kevin Wayne's slides, there are important applications for this problem.)

Like sorting, counting inversions done naively by brute force (i.e., trying every pair) would result in $O(n^2)$ comparisons (and time complexity) for an array of size n. We assme that everyone is familiar with the merge sort algorithm that sorts in $O(n \log n)$ comparisons.

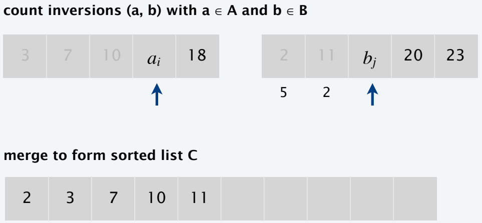
An elegant way to count inversions in $O(n \log n)$ comparisons is to generalize the problem to sort the array (using merge sort) and count inversions while doing the merging.
Here follows (some of) the relevant slides.

Counting inversions by *sort and count*

Counting inversions: how to combine two subproblems?

Count inversions (a, b) with $a \in A$ and $b \in B$, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
- If $a_i < b_j$, then a_i is not inverted with any element left in B.
- If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.



count inversions (a, b) with a ∈ A and b ∈ B
3
7
10
a_i
18
2
11
b_j
20
23
5
2
merge to form sorted list C
2
3
7
10
11

Counting inversions by *sort and count*

Counting inversions: divide-and-conquer algorithm implementation

Input. List L.

Output. Number of inversions in L and L in sorted order.

```
SORT-AND-COUNT($L$)

IF (list $L$ has one element)
    RETURN $(0, L)$.

Divide the list into two halves $A$ and $B$.
$(r_A,\ A) \leftarrow$ SORT-AND-COUNT($A$).          $\longleftarrow T(n/2)$
$(r_B,\ B) \leftarrow$ SORT-AND-COUNT($B$).          $\longleftarrow T(n/2)$
$(r_{AB},\ L) \leftarrow$ MERGE-AND-COUNT($A, B$).   $\longleftarrow \Theta(n)$

RETURN $(r_A + r_B + r_{AB},\ L)$.
```

Counting inversions by *sort and count*

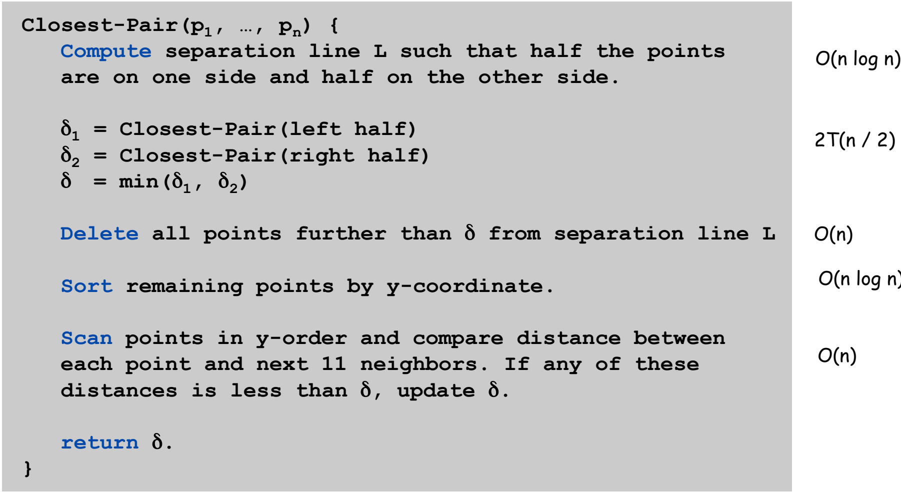
Counting inversions: divide-and-conquer algorithm analysis

Proposition. The sort-and-count algorithm counts the number of inversions in a permutation of size n in $O(n \log n)$ time.

Pf. The worst-case running time $T(n)$ satisfies the recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Closest pair in $\mathbb{R}^2$ from Kevin Wayne's slides



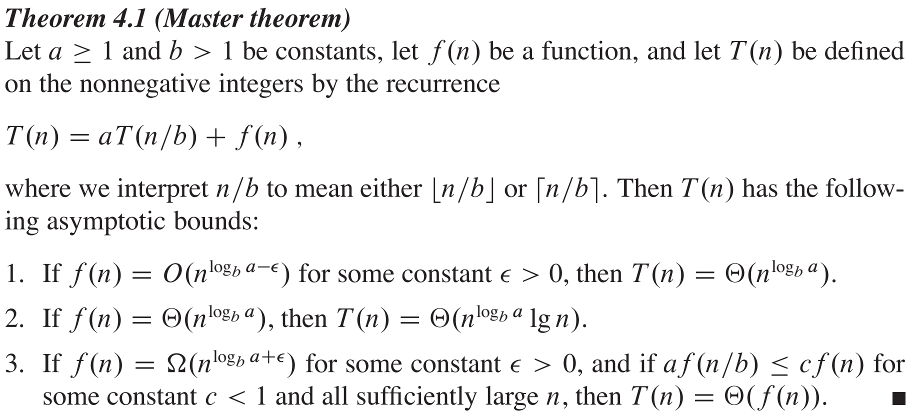
```
Closest-Pair(p_1, …, p_n) {
   Compute separation line L such that half the points      O(n log n)
   are on one side and half on the other side.

   δ_1 = Closest-Pair(left half)                            2T(n / 2)
   δ_2 = Closest-Pair(right half)
   δ  = min(δ_1, δ_2)

   Delete all points further than δ from separation line L  O(n)

   Sort remaining points by y-coordinate.                    O(n log n)

   Scan points in y-order and compare distance between      O(n)
   each point and next 11 neighbors. If any of these
   distances is less than δ, update δ.

   return δ.
}
```

Recurrences describing divide and conquer algorithms

- From previous courses (and previous examples), we have seen the recurrences describing the divide and conquer algorithms for counting inversions and closest points in $\mathbb{R}^2$. Namely $T(n) = 2T(n/2) + O(n)$ and $T(1) = O(1)$ so that $T(n) = O(n \log n)$.
- The next two divide and couquer examples examples result in recurrences of the form $T(n) = aT(n/b) + f(n)$ for $a > b$ where $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$. so that $T(n) = n^{\log_b a}$.
- These are all cases of the so called *master theorem*

The master theorem

Here is the master theorem as it appears in CLRS



Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Karatsuba's interger multiplication from Kevin Wayne's slides

Karatsuba Multiplication

To multiply two n-bit integers a and b:

- Add two $\frac{1}{2}n$ bit integers.
- Multiply three $\frac{1}{2}n$-bit integers, recursively.
- Add, subtract, and shift to obtain result.

$$
\begin{aligned}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= 2^{n} \cdot a_1 b_1 + 2^{n/2} \cdot \left(a_1 b_0 + a_0 b_1\right) + a_0 b_0 \\
&= 2^{n} \cdot \underset{\textcircled{1}}{a_1 b_1} + 2^{n/2} \cdot \left(\underset{\textcircled{2}}{(a_1 + a_0)(b_1 + b_0)} - \underset{\textcircled{1}}{a_1 b_1} - \underset{\textcircled{3}}{a_0 b_0} \right) + \underset{\textcircled{3}}{a_0 b_0}
\end{aligned}
$$

Theorem. [Karatsuba-Ofman 1962] Can multiply two n-bit integers in $O(n^{1.585})$ bit operations.

Strassen's $n \times n$ fast matrix multiplication

This algorithm is described in all the texts. While there is some question as to when fast matrix multiplicaton (Strassen's and subsequent asymptotically faster algorithms) has practical application, it plays a seminal role in theoretical computer science.

Strassen's $n \times n$ fast matrix multiplication

This algorithm is described in all the texts. While there is some question as to when fast matrix multiplicaton (Strassen's and subsequent asymptotically faster algorithms) has practical application, it plays a seminal role in theoretical computer science.

The standard method to multiply two $n \times n$ matrices requires $O(n^3)$ scalar $(+, \cdot)$ operations. There were conjectures (and a published false proof!) that *any algorithm* for matrix multipication requires $\Omega(n^3)$ scalar operations.

Strassen's $n \times n$ fast matrix multiplication

This algorithm is described in all the texts. While there is some question as to when fast matrix multiplicaton (Strassen's and subsequent asymptotically faster algorithms) has practical application, it plays a seminal role in theoretical computer science.

The standard method to multiply two $n \times n$ matrices requires $O(n^3)$ scalar $(+, \cdot)$ operations. There were conjectures (and a published false proof!) that *any algorithm* for matrix multipication requires $\Omega(n^3)$ scalar operations.

Why would you make such a conjecture? And why is this such a seminal result?

Strassen's $n \times n$ fast matrix multiplication

This algorithm is described in all the texts. While there is some question as to when fast matrix multiplicaton (Strassen's and subsequent asymptotically faster algorithms) has practical application, it plays a seminal role in theoretical computer science.

The standard method to multiply two $n \times n$ matrices requires $O(n^3)$ scalar $(+, \cdot)$ operations. There were conjectures (and a published false proof!) that *any algorithm* for matrix multipication requires $\Omega(n^3)$ scalar operations.

Why would you make such a conjecture? And why is this such a seminal result?

Theorem (Strassen): Matrix multiplcation (over any ring) can be realized in $O(n^{\log_2 7})$ scalar operations.

Furthermore, Strassen shows that matrix inversion for a non-singluar matrix reduces (and is equivalent) to matrix multiplication.

Strassen's matrix multiplication continued

The high level idea is conceptually simple. The method is based on Strassen's *insightful* (and not simple) discovery that 2×2 matrix multiplication can be realized in 7 (not 8) non-commutative multiplications and 18 additions. (Note: the number of additions will only impact the hidden constant in the "big O" notation and not the matrix mutliplication exponent.)

Strassen's matrix multiplication continued

The high level idea is conceptually simple. The method is based on Strassen's *insightful* (and not simple) discovery that 2×2 matrix multiplication can be realized in 7 (not 8) non-commutative multiplications and 18 additions. (Note: the number of additions will only impact the hidden constant in the "big O" notation and not the matrix mutliplication exponent.)

The insights into the 2×2 method are beyond the scope of this course so lets just see how the $n \times n$ result follows. Without loss of generality, let $n = 2^k$ for some k.

Strassen's matrix multiplication continued

The high level idea is conceptually simple. The method is based on Strassen's *insightful* (and not simple) discovery that 2×2 matrix multiplication can be realized in 7 (not 8) non-commutative multiplications and 18 additions. (Note: the number of additions will only impact the hidden constant in the "big O" notation and not the matrix mutliplication exponent.)

The insights into the 2×2 method are beyond the scope of this course so lets just see how the $n \times n$ result follows. Without loss of generality, let $n = 2^k$ for some k.

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

Figure: Viewing an $n \times n$ matrix as four $n/2 \times n/2$ matrices

Strassen's matrix multiplication continued

Since matrices are a non commutative ring (i.e., matrix multiplication is not commutative), the 2×2 result can be applied so that an $n \times n$ mutilpication can be realized in 7 $n/2 \times n/2$ matrix multiplications (and 18 matrix additions).

Strassen's matrix multiplication continued

Since matrices are a non commutative ring (i.e., matrix multiplication is not commutative), the 2×2 result can be applied so that an $n \times n$ mutilpication can be realized in 7 $n/2 \times n/2$ matrix multiplications (and 18 matrix additions).

Since matrix addition uses only $O(n^2)$ scalar operations, the number $T(n)$ of scalar opeations is determined by the recurrence :

$T(n) = 7 * T(n/2) + O(n^2)$ with $T(1) = 1$.

This implies the stated result that $T(n) = O(n^{\log_2 7})$.

Other examples of the master theorem

There are a few other cases of the master theorem that often occur. We will assume that $T(1) = O(1)$.

- The recurrence $T(n) = 2T(n/2) + O(1)$ and $T(1) = O(1)$ implies $T(n) = O(n)$
  A not so useful example: finidng the maximum element in an unsorted list.
  Somewhat perhaps more useful is to find the minimum and maximum element in $\lceil \frac{3n}{2} \rceil$ comparisons.
- The recurrence $T(n) = T(n/b) + O(n)$ for $b > 1$ implies $T(n) = O(n)$.
  For an example, see exercise 4-5 in CLRS (page 109). Later we will discuss how to find the median element in an unsorted list in $O(n)$.
- The recurrence $T(n) = T(n/b) + O(1)$ for $b > 1$ implies $T(n) = O(\log n)$.
  The standard binary search in a sorted list is a typical example.

What has to be proven?

In general, when analyzing an algorithm, we have to do two basic things:

1. Prove coorectnesss; that is, that the algorithm realizes the required problem specification. For example, for Strassen's matrix multiplication, it must be shown that the output matrix is the product of the two input matrices. For the closest pair problem we need to prove that the output is the closest pair of points; that is, that an optimal solution was obtained. Later, when considering approximation algorithms, we need to prove that the algorithm produces a feasible solution within some factor of an optimal solution.
2. Analyze the complexity of the algorithm in terms of the input parameters of the problem. For us, we will mainly be interested in the (sequential) time of the algorithm as a function $T()$ of the "size" of the input representation. For example, in $n \times n$ matrix multiplication, the usual measure of "size" is the size n of the matrices. But if were considering the multiplication of $A_{m,n} \cdot B_{n,p}$, we would want to analyze the time complexity $T(m, n, p)$.

More on compelxity analysis

- For divide and conquer, we analyze the complexity by establishing a recurrence and then solving that recurrence. For us, the recurrences are usually solved by the master theorem. In fact, if we know the desired time bound, we can sometimes guess a suitable recurrence which may suggest a framework for a possible solution.
- There are other important complexity measures besides sequential time, including parallel time (if we are in a model of parallel computation) and memory space.
- For much of our algorithm analysis (as in all the previous examples except integer multiplication), we are assumiung a random access model and counting the number of machine operations (e.g., comparisons, arithmetic operations) ignoring representation issues (e.g., the number of bits or digits in the matrix entries, or the representation of the "real numbers" in the closest pair problem). For interger multiplcation, we measured the "size" of the input representation in terms of the number of bits or digits of the two numbers.

Looking ahead to complexity theory

When we get to complexity theory, the standard measure is the nunber of bits (or digits) in the representation of the inputs. In particular, when discussing the "P vs NP issue and NP completeness, we assume that we have string (o0ver some finite alphabet) representation of the inputs. (We will not usually have to worry about the size of the output.)

This makes sense as for example, in integer factoring (the basis for RSA cryptography) it would not make good sense to measure complexity of the value x of the number being factored but rather we need to measure compexity as a function of the number of bits (or digits) to represent x. Why?

Looking ahead to complexity theory

When we get to complexity theory, the standard measure is the nunber of bits (or digits) in the representation of the inputs. In particular, when discussing the "P vs NP issue and NP completeness, we assume that we have string (o0ver some finite alphabet) representation of the inputs. (We will not usually have to worry about the size of the output.)

This makes sense as for example, in integer factoring (the basis for RSA cryptography) it would not make good sense to measure complexity of the value x of the number being factored but rather we need to measure compexity as a function of the number of bits (or digits) to represent x. Why?

This distinction (between the value of say an integer and its representation length) will also become important when we discuss the knapsack problem in the context of dynamic programming.

Computing the median and selecting the i^{th} smallest element

As a special case of the third part of the master theorem, the recurrence $T(n) = T(n/b) + O(n)$ implies $T(n) = O(n)$.

We would like to design an algorithm for computing the median in an unsorted list of n elements using only $O(n)$ comparisons. (We can rather more naively simply sort using time $O(n \log n)$ comparisons.) There is a deterministic algorithm based on this recurrence but it is simpler to consider a randomized algorithm whose *expected time* is $O(n)$.

It turns out that in trying to come up with an appropriate divide and conquer algorithm for the median, it is "more natural" to consider the more general problem of selecting the i^{th} smallest (or largest) element.

The following approach is based on the partitioning idea in randomized quicksort, and then using the intended recurrence to motivate the selection algorithm (see sections 7.1 and 9.2 in CLRS).

In-place partition

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

Figure: In-place parition

Randomized quicksort

```
RANDOMIZED-PARTITION($A, p, r$)
1  $i$ = RANDOM($p, r$)
2  exchange $A[r]$ with $A[i]$
3  return PARTITION($A, p, r$)
```

The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

```
RANDOMIZED-QUICKSORT($A, p, r$)
1  if $p < r$
2      $q$ = RANDOMIZED-PARTITION($A, p, r$)
3      RANDOMIZED-QUICKSORT($A, p, q - 1$)
4      RANDOMIZED-QUICKSORT($A, q + 1, r$)
```

Figure: Quicksort

Randomized selection of i^{th} smallest in $A[p], \ldots, A[r]$

```
RANDOMIZED-SELECT$(A, p, r, i)$
if $p == r$
    return $A[p]$
$q = $ RANDOMIZED-PARTITION$(A, p, r)$
$k = q - p + 1$
if $i == k$                // the pivot value is the answer
    return $A[q]$
elseif $i < k$
    return RANDOMIZED-SELECT$(A, p, q - 1, i)$
else return RANDOMIZED-SELECT$(A, q + 1, r, i - k)$
```

Figure: Randomized selection

The DFT and FFT

One of the most important procedures in signal processing is the use of the Fast Fourier Transform (FFT). The FFT is an algorithm for computing the Discrete Fourier Transform (DFT).

To define the DFT, we need the concept of a primitive n^{th} root of unity in (say) the complex plane. Namely, ω is a primitive n^{th} root of unity if

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

a
For simpliity (but now this is a restriction) assume $n = 2^k$ for some integer $k \geq 1$.

- If ω is a primitive n^{th} root of unity then ω^2 is a primtive $(n/2)^{th}$ root of unity.
- If ω is a primitive n^{th} root of unity then ω^{-1} is a primitive primitive n^{th} root of unity.

Defining the DFT

The DFT_n is defined as a matrix vector multiplcation $\mathbf{y} = V \cdot \mathbf{a}$ where V is a Vandermond matrix with respect to the primitive n^{th} root of unity ω_n.

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

Figure: The n^{th} order DFT_n where ω_n is a primitive n^{th} root of unity

Note: We assuming that $n = 2^k$ for some integer $k \geq 0$. For some applications of the FFT, we cannot make this assumption and then a more substantial development is needed. However, the DFT is well defined for all n.

The FFT; Computing the DFT_n in $O(n \log n)$ arithmetic operations

In general, computing a matrix vector product $G_{n\times n} \cdot \mathbf{z}$ requires $\Theta(n^2)$ arithmetic operations. But the Vandermonde matrix is obviously a special matrix.

Observation 1: We can think of the matrix vector product $V \cdot \mathbf{a}$ as the evaluation of the polynomial $\sum_{i=0}^{n-1} a_i x^i$ at the n points $x_i = \omega_n^i$ for $i = 0, \ldots, n-1$.

Observation 2: If ω_n is a primitve n^{th} root of unity then $\omega_{n/2} = \omega_n^2$ is a primitive $(\frac{n}{2})^{th}$ root of unity and $\omega_{n/2}^{n/2} = 1$. (Recall we are assuming $n = 2^k$.)

To hopefully simplify notation, let $\omega_{n,j}$ denote ω_n^j so that our goal is to evaluate $a(x)$ at the n points $\omega_{n,j}$ $(j = 0, 1, \ldots, n-1)$.

The FFT continued: the basic obervation

We can express polynomial evaluation as follows:
$\sum_{i=0}^{n-1} a_i x^i = \sum_{i=0}^{\frac{n}{2}-1} a_{2i} x^{2i} + x \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} x^{2i}$ (i.e. even terms + odd terms)

The FFT continued: the basic obervation

We can express polynomial evaluation as follows:
$\sum_{i=0}^{n-1} a_i x^i = \sum_{i=0}^{\frac{n}{2}-1} a_{2i} x^{2i} + x \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} x^{2i}$ (i.e. even terms + odd terms)
Therefore,

$$\begin{aligned}\sum_{i=0}^{n-1} a_i \omega_{n,j}^i &= \sum_{i=0}^{\frac{n}{2}-1} a_{2i} \omega_{n,j}^{2i} + \omega_n \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} \omega_{n,j}^{2i} \\ &= \sum_{i=0}^{\frac{n}{2}-1} a_i \omega_{n/2,j}^i + \omega_{n,j} \sum_{i=0}^{\frac{n}{2}-1} a_i \omega_{n/2,j}^i \quad \text{when } x = \omega_{n,j}\end{aligned}$$

The FFT continued: the basic obervation

We can express polynomial evaluation as follows:
$\sum_{i=0}^{n-1} a_i x^i = \sum_{i=0}^{\frac{n}{2}-1} a_{2i} x^{2i} + x \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} x^{2i}$ (i.e. even terms + odd terms)
Therefore,

$$\begin{aligned}\sum_{i=0}^{n-1} a_i \omega_{n,j}^i &= \sum_{i=0}^{\frac{n}{2}-1} a_{2i} \omega_{n,j}^{2i} + \omega_n \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} \omega_{n,j}^{2i} \\ &= \sum_{i=0}^{\frac{n}{2}-1} a_i \omega_{n/2,j}^i + \omega_{n,j} \sum_{i=0}^{\frac{n}{2}-1} a_i \omega_{n/2,j}^i \quad \text{when } x = \omega_{n,j}\end{aligned}$$

It follows that by this rearrangement, the computation of DFT_n reduces to two instances of the $DFT_{n/2}$ + n scalar multiplications + n scalar additions.
Therefore $T(n) = 2T(n/2) + O(n)$

The DFT/FFT is discussed in CLRS, section 30.2. We will elaborate a little more in the next lecture.

Solving recurrences

Once we know (or have good intuition) for what the solution of a recurrence is, we can usually verify the solution by induction.

What should we do when we don't yet know what to expect?

Solving recurrences

Once we know (or have good intuition) for what the solution of a recurrence is, we can usually verify the solution by induction.

What should we do when we don't yet know what to expect?
Well, of course, one can look up many examples of recurrences say on the internet and that often works.

Solving recurrences

Once we know (or have good intuition) for what the solution of a recurrence is, we can usually verify the solution by induction.

What should we do when we don't yet know what to expect?
Well, of course, one can look up many examples of recurrences say on the internet and that often works.
And if that doesn't work?

Solving recurrences

Once we know (or have good intuition) for what the solution of a recurrence is, we can usually verify the solution by induction.

What should we do when we don't yet know what to expect?
Well, of course, one can look up many examples of recurrences say on the internet and that often works.
And if that doesn't work?
Perhaps the most common approach is to start to unwind the recurrence to see if it suggests a pattern . The CLRS text calls this the recursion tree method. (Of course, "seeing this pattern" is somewhat of an art form and some recurrences are not easy to solve.)

In most cases, if we are only intersted in asymptotic bounds, then we can assume say that n has some special form such as $n = 2^k$ for some $k \geq 0$ and that eliminates the concern for handling floors and ceilings.

Solving recurrences

Once we know (or have good intuition) for what the solution of a recurrence is, we can usually verify the solution by induction.

What should we do when we don't yet know what to expect?
Well, of course, one can look up many examples of recurrences say on the internet and that often works.
And if that doesn't work?
Perhaps the most common approach is to start to unwind the recurrence to see if it suggests a pattern . The CLRS text calls this the recursion tree method. (Of course, "seeing this pattern" is somewhat of an art form and some recurrences are not easy to solve.)

In most cases, if we are only intersted in asymptotic bounds, then we can assume say that n has some special form such as $n = 2^k$ for some $k \geq 0$ and that eliminates the concern for handling floors and ceilings.

An easy and common recureence to derive this way is
$T(n) = 2T(n/2) + O(n)$.

The recursion tree for $T(n) = 2T(n/2) + cn$

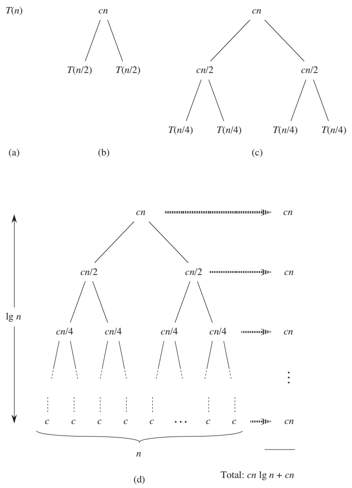


Figure: Figure 2.5 in CLRS; the recursion tree for the reurrence $T(n) = 2T(n/2) + cn$