CS445 – Introduction to Algorithms

• **Course Staff**
  
  - Alon Efrat,
  - Jacob Miller
  - Fan Lou
  - Prithula Hridi

• **Webpages**
  
  - Course webpage – google doc (reach via my homepage)
  - Use D2L to reach recordings of lectures (Panopto), calendar
  - Use Gradescope to submit his and view feedback
  - Use Piazza for course communication, discussions and announcements.
  - Use Overleaf to view assignments.
Homeworks workflow. Collaboration vs Cheating

- Alg: Once a homeworks is published
  - Read questions
  - If needed, re-watch lectures (Alon and Others) online,
  - Thinks really hard. Discover what does **not** work and why
  - Meet your peers and discuss and does/does not work and why?
  - Write Solutions yourself.

- Diverging from this algorithm might improve your hw grade but is likely to impact your exams grades (not to mention ethical issues, honor code etc).

- Homework's rules.
  - Collaborations ++. Brainstorming in **small** groups
  - Give credit. Specify your contribution to each solution (in %).
  - Sharing text is cheating.
CS445 - Regulation, Bureaucracy

1. Grading Scheme (midterm vs. final)
2. Textbooks
3. Video recording
4. Web Resources
5. Prerequisites (course is mostly self contained, but harder if you did not pass cs345.
   I. Post are for clarifications.
   II. Be careful not to share any hints in your posts
      Eg. “are we allowed to use Quicksort for the solution of hw3 Q7” is a violation of code of conduct, considered cheating, and could get you blocked from piazza.
   I. If you have any doubts, send a private message.
7. Attendance - strongly recommended.
   I. Active learning - your webcam should be on during active learning (talk to me if there are any technical difficulties).
1. **Textbook**

- **CLRS**
- **Kleinberg & Tardos**
- **Sanjoy Dasgupta**
- **Lewis Denenberg**

*Course slides*
Introduction to Algorithms

In this course, we will discuss problems, and algorithms for solving these problems.

There are so many algorithms – why focus on the ones in the syllabus?
Why study algorithms and performance?

- Performance often draws the line between what is feasible and what is impossible.

- Algorithmic mathematics provides a language for talking about program behavior.
  
  - (e.g., by using big-$O$ notation.

  - Will see lots of `big-$O$’s of quantities you might have not seen before:

    - (CPU, Space, I/O, parallel steps, GPU)

- In real life, many algorithms, though different from each other, fall into one of several paradigms (discussed shortly).

- These paradigms can be studied, and applied for new problems.
Why these algorithms (cont.)

1. Main paradigms:
   a) Greedy algorithms
   b) Divide-and-Conquers
   c) Dynamic programming
   d) Brach-and-Bound (mostly in AI)
   e) Etc etc.

2. Other reasons:
   a) Relevance to many areas:
      • E.g., networking, internet, search engines...
   b) Coolness
Other goals of the course

• Knowing when running time counts, and what to do when it does

• Magic of randomness and sampling
**O-notation**

We say that \( T(n) = O(g(n)) \) iff there exists positive constants \( c_1 \), and \( n_0 \) such that \( 0 \leq T(n) \leq c_1 g(n) \) for all \( n \geq n_0 \)

Usually \( T(n) \) is running time, and \( n \) is size of input

- We shouldn’t ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- **Asymptotic analysis** is a useful tool to help to structure our thinking.
\( \Omega \)-notation

**Math:**

We say that \( T(n) = \Omega(g(n)) \) iff there exists positive constants \( c_2 \), and \( n_0 \) such that 
\[
0 \leq c_1 g(n) \leq T(n) \quad \text{for all } n \geq n_0
\]

**Engineering:**

- Drop low-order terms; ignore leading constants.
- Example: \( 3n^3 + 90n^2 - 5n + 6046 = \Omega(n^3) \)
We say that \( T(n) = \Theta(g(n)) \) iff there are positive constants \( c_1, c_2, \) and \( n_0 \) such that
\[
0 \leq c_1 g(n) \leq T(n) \leq c_1 g(n)
\]
for every \( n \), provide that \( n \geq n_0 \)
in other words, we could say that
\[
T(n) = \Theta(g(n))
\]
iff it is true that
\[
T(n) = O(g(n)) \quad \text{and} \quad T(n) = \Omega(g(n)).
\]
For example, for every size \( n \) of an input array, bubble sort, insertion sort and swap-sort will never needs more than \( n^2 \) operations (up to a constant).

So their running time is \( O(n^2) \).

On the other hand, we can find an input (one is enough) that causes their running time to be no less than \( n^2 \). So their running time is also \( \Omega(n^2) \).

Putting it together, their running time is \( \Theta(n^2) \)
Notation - cont

So if \( T(n) = \Theta(n^2) \) then we are also sure that
\[ T(n) = O(n^3) \] and that
\[ T(n) = O(n^{3.5}) \] and
\[ T(n) = O(2^n) \]

But it might or might not be true that \( T(n) = O(n^{1.5}) \).

However, if \( T(n) = \Omega(n^2) \) then it is **not** true that
\[ T(n) = O(n^{1.5}) \]

**Big difference between \( O \) and \( \Omega \):** we can talk about \( \Omega \) of a **problem**
(that is, any algorithm that solves this problem takes \( \Omega(\text{something}) \)

Eg. Sorting takes \( \Omega(n \log n) \)
Examples 3

What is the running time of this code (as a function of \( n \))?

1. Read(n);
2. \( k=1 \);
3. while( \( k \leq n \) )
4. \( k=2k \);

• We know that each iteration takes \( O(1) \) times. Need to find the number time line 3 is executed.
  • After the first iteration \( k=2=2^1 \)
  • After the 2nd iteration \( k=4=2^2 \)
  • After the 3rd iteration \( k=8=2^3 \)
  • ....
  • After the \( i \)'th iteration \( k=2^i \)

Cheatsheet:

- \( \log(ab)=\log(a)+\log(b) \)
- \( \log(a^b)=b\log(a) \)
- \( \log_a(x)=\frac{\log_b(x)}{\log_b(a)} \)
- \( x \leq y \implies \log_2(x) \leq \log_2(y) \)

Let's count the number \( j \) of times that the condition of line 3 was checked and yield true.

• If the condition is true, then \( k \leq n \). But \( k=2^j \). So \( k = 2^j \leq n \).
• Taking \( \log_2 \) from both sides, we have that
  \[ \log_2 k = \log_2(2^j) \leq \log_2(n) \] or...
  \[ \log_2(2^j) = j \log_2 2 = j \leq \log_2(n) \] or..
  \[ j=O(\log_2 n). \quad T(n)=O(\log n) \]
• Homework: Prove \( T(n)=\Theta(\log n) \)
read(n);  
for(i=1 ; i < n ; i++)  
  for(j=i ; j <n ; j += i)  
    print("*") ;

• Time Complexity Analysis – first approach:  
  • The outer loop (on i) runs exactly n-1 times  
  • The inner loop (on j) runs O(n) times.  
  • Together \( T(n) = O(n^2) \).

• More “sensitive” analysis:  
  • For \( i=1 \) we run through \( j=1,2,3,4...n \), total \( n \) times.  
  • For \( i=2 \) we run through \( j=2,4,6,8,10...n \), total \( n/2 \) times.  
  • For \( i=3 \) we run through \( j=3,6,9,12...n \), total \( n/3 \) times.  
  • For \( i=4 \) we run through \( j=4,8,12,16...n \), total \( n/4 \) times.  
  • For \( i=n \) we run through \( j=n \), total \( n/n=1 \) times.

• Summing up:  
\[
T(n) = n + n/2 + n/3 + n/4 + ... n/n = n(1 + 1/2 + 1/3 + 1/4 + ... 1/n) \approx n \ln n
\]

Harmonic Sum
Example 5

Pay attention -
very relevant to this course

```c
read(n);  a=0.5
while( n>1) {
    For( j=1; j<n ; j++ ) print(“*”)
    n=a*n ;
}
```

• The **first** time the outer loop is called, the “print” is called $n$ times.
• The **2nd** time the outer loop is called, the “print” is called $an$ times.
• The **3rd** time the outer loop is called, the “print” is called $a^2n$ times…
• The **k’th** time the outer loop is called, the “print” is called $a^k n$ times

• Let $t$ be the number of iterations of the outer loop. Then the total time

$$= n + an + a^2n + a^3n + \ldots a^tn = n(1 + a + a^2 + a^3 + \ldots a^t) < n(1 + a + a^2 + a^3 + \ldots a^t + \ldots) = n / (1-a) = O(n).$$

• Same analysis holds for any $a<1$

Recall: $1+a+a^2+\ldots+a^t= (1-a^{t+1})/(1-a)$.
If $a<1$ then $1+a+a^2+\ldots+a^t+\ldots = 1/(1-a)$
More about \( \Omega(\ ) \)

Sometimes we would talk about a lower bound on the running time of a **specific algorithms**

E.g. The insertion sort might take \( \Omega(n^2) \) for some input

Sometimes we would talk about a lower bound on the running time of a **problem**

E.g.

1. Any algorithms that reads all the input (for any problem) requires \( \Omega(n) \) time.
2. Any algorithm that stores all the data requires \( \Omega(n) \) space.
3. Any algorithm that sort \( n \) keys requires \( \Omega(n \log n) \)
   (disclaimer – could be better if we make some assumptions about the keys or the model. Usually
   • Sorting **sort integers** takes \( \Omega(n) \) (how?)
   • Sorting **floats** takes \( \Omega(n \log n) \)