**CS445**

*Introduction to Algorithms*

Instructor: **Alon Efrat**

Some slides are courtesy of
Piotr Indyk and Carola Wenk

---

### Polices

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.
   1. Post are for clarifications about the questions.
   2. Be careful not to share any hints in your posts
      Eg. "Could we use Quicksort for the solution of hw3 Q7" is a violation of code of conduct and should not be asked publically.
      "Does anybody else feels that..."
7. Attendance

---

### Homeworks

- Alg: Once homeworks are published (and unless stated otherwise)
  - Read questions
  - If needed, re-watch lectures (Alon and Others) online,
  - Thinks really hard. Discover what does not work and why
  - Meet your peers (no more than two of them) and and discuss.
  - 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 (max 4).
  - 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.
   1. Post are for clarifications about the questions.
   2. Be careful not to share any hints in your posts
      Eg. "Could we use Quicksort for the solution of hw3 Q7" is a violation of code of conduct and should not be asked publically.
      "Does anybody else feels that..."
7. Attendance
Could we use screens in the classroom?

1. Screens - allowable, but your focus should be centered at the lecture.
2. Keep regular eye contact with me.
3. If you hear ”no screens for you today” it is not an insult.

Class participations

• Class participations might effect your grade positively.
• Class participation is expressed via
  1. Asking questions in the classroom
  2. Answering lecturer’s questions in the classroom
  3. Office hours
  4. Piazza discussions

Why these particular algorithms ??

sponsor

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 these algorithms (cont.)

− Relevance to many areas:
  • E.g., networking, search engines, ...
− Coolness
− Demonstrate important non-trivial ideas that could be used as building blocks to solutions to other problems that are not discussed in the course (more on it next slide)
− Main Paradigms
Why these algorithms (cont.)

Main paradigms:
- a) Greedy algorithms
- b) Divide-and-Conquer
- c) Dynamic programming
- d) Randomized incremental
- e) Branch-and-Bound (mostly in AI)
- f) Etc etc.

Next – Some Sorting Algorithms

- Good occasion to go over two types of sorting algorithms (you are suppose to know more from CSc345)
- Practice asymptotic time complexity notations.

The problem of sorting

*Input:* sequence \(\langle a_1, a_2, \ldots, a_n \rangle\) of numbers.

*Output:* permutation \(\langle a'_1, a'_2, \ldots, a'_n \rangle\) such that \(a'_1 \leq a'_2 \leq \ldots \leq a'_n\).

**Example:**
*Input:* 8 2 4 9 3 6
*Output:* 2 3 4 6 8 9

Insertion sort

**Invariants:**
- the subarray \(A[1...j-1]\) is sorted

*Consider \(A[j]=9\). Obviously violating the invariant. (Not in the correct place.)*
*Need to make room for 9 somewhere to its left.*
*Need to shift all elements right, starting from 10.*
Insertion sort

```
INSERTION-SORT (A, n)  //input: A[1..n]
for j ← 2 to n  //outer loop
do  key ← A[j]
i ← j - 1
while i > 0 and A[i] > key  //inner loop
do  { A[i+1] ← A[i]  
i ← i - 1 }
A[i+1] = key
```

Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

2 8 4 9 3 6
Example of insertion sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6
Example of insertion sort

```
8  2  4  9  3  6
2  8  4  9  3  6
2  4  8  9  3  6
2  4  8  9  3  6
2  3  4  8  9  6
done
```
Running time

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input \( n \).
- Seek upper bounds on the running time \( T(n) \) for the input size \( n \), because everybody likes a guarantee.

Machine-independent time

What is insertion sort’s worst-case time?
- It depends on the speed of our computer:
  - relative speed (on the same machine),
  - absolute speed (on different machines).

**Big Idea:**
- Ignore machine-dependent constants.
- Look at growth of \( T(n) \) as \( n \to \infty \).

**“Asymptotic Analysis”**

\[ O \text{-notation} \]

Let \( T(n) \) denote (say) the running time of the insertion sort algorithm on an input of \( n \) keys. We say that \( T(n) = O(g(n)) \) iff there exists positive constants \( c_1 \) and \( n_0 \) such that

\[ 0 \leq T(n) \leq c_1 g(n) \quad \text{for all } n \geq n_0. \]

In this course, in most cases, \( g(n) \) is one of the functions
- \( \log n \)
- \( n \log n \)
- \( n^2 \)

**Engineering:**

- Drop low-order terms; ignore leading constants.
- Example: \( 3n^3 + 90n^2 - 5n + 6046 = O(n^3) \)
**Θ-notation**

**Math:**
we say that \( T(n) = \Theta(g(n)) \) if and only if there exist positive constants \( c_1, c_2, \) and \( n_0 \) such that \( 0 \leq c_1 g(n) \leq T(n) \leq c_2 g(n) \) for all \( n \geq n_0 \)

**Engineering:**
• Drop low-order terms; ignore leading constants.
• Example: \( 3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3) \)

---

**O-notation - cont**

So if \( T(n) = O(n^2) \) then we are also sure that:
- \( T(n) = O(n^3) \)
- \( T(n) = O(n^{1.5}) \)
- \( T(n) = O(2^n) \)

---

**Asymptotic performance**

When \( n \) gets large enough, a \( \Theta(n^2) \) algorithm always beats a \( \Theta(n^3) \) algorithm.

- 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.

---

**Insertion sort analysis**

**Worst case:** Input reverse sorted.

\[
T(n) = 2c + 3c + 4c + \ldots + cn = cn(n-1)/2
\]

\( c \) is a constant (the time to perform a basic operation)

- Is insertion sort a fast sorting algorithm?
  - Moderately so, for small \( n \).
  - Not at all, for large \( n \).
Merge sort
(divide-and-conquer algorithm)

MERGE-SORT $A[1 \ldots n]$
1. If $n = 1$, done.
2. Recursively sort $A[1 \ldots \lfloor n/2 \rfloor]$ and $A[\lceil n/2 \rceil + 1 \ldots n]$.
3. “Merge” the 2 sorted lists.

Key subroutine: MERGE

Merging two sorted arrays

20 12
13 11
7 9

2 1

Merging two sorted arrays

20 12
13 11
7 9

2 1

1
Merging two sorted arrays

20 12 7 9 2 1
13 11 12 2 2

Merging two sorted arrays

20 12 7 9 2 1
13 11 12 2 2

Merging two sorted arrays

20 12 7 9 2 1
13 11 12 2 2

Merging two sorted arrays

20 12 7 9 2 1
13 11 12 2 2

Merging two sorted arrays

20 12 7 9 2 1
13 11 12 2 2

Merging two sorted arrays

20 12 7 9 2 1
13 11 12 2 2

Merging two sorted arrays

20 12 7 9 2 1
13 11 12 2 2

Merging two sorted arrays

20 12 7 9 2 1
13 11 12 2 2
Merging two sorted arrays

1. Merge the two arrays by comparing elements from both arrays.
2. Place the smaller element in the merged array.
3. Move the pointers of both arrays to the next element.
4. Repeat steps 1-3 until all elements are merged.

Example:

Array 1: 1, 2, 7
Array 2: 1, 9
Merged Array: 1, 1, 2, 7, 9
Merging two sorted arrays

Time = $\Theta(n)$ to merge a total of $n$ elements (linear time).

Analyzing merge sort

1. If $n = 1$, done.
2. Recursively sort $A[1 \ldots \lfloor n/2 \rfloor]$ and $A[\lceil n/2 \rceil + 1 \ldots n]$.
3. “Merge” the 2 sorted lists

Sloppiness: Should be $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)$, but it turns out not to matter asymptotically.

Recurrence for merge sort

$T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1; \\
2T(n/2) + \Theta(n) & \text{if } n > 1.
\end{cases}$

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small $n$, but only when it has no effect on the asymptotic solution to the recurrence.
- CLRS provides several ways to find a good bound on $T(n)$. 
Recursion tree
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.
Recursion tree
Solve \( T(n) = 2T(n/2) + cn \), where \( c > 0 \) is constant.

\[
\begin{align*}
T(n) &= 2T(n/2) + cn \\
&= 2(2T(n/4) + cn/2) + cn \\
&= 4T(n/4) + 2cn/2 + cn \\
&= 4(2T(n/8) + cn/4) + 2cn/2 + cn \\
&= 8T(n/8) + 4cn/4 + 2cn/2 + cn \\
&= \ldots \\
&= 2^h T(n/2^h) + \sum_{i=0}^{h-1} 2^i cn/2^i \\
&= 2^h T(1) + \sum_{i=0}^{h-1} 2^i cn/2^i \\
&= \Theta(1) + \sum_{i=0}^{h-1} cn/2^i \\
&= \Theta(1) + \frac{cn}{2} \left(1 - \frac{1}{2^h}\right) \\
&= \Theta(n) \\
\end{align*}
\]
Recursion tree
Solve \( T(n) = 2T(n/2) + cn \), where \( c > 0 \) is constant.

\[
\begin{align*}
  h &= \log n \quad \text{with} \\
  \Theta(1) &\quad \text{and} \\
  \Theta(n^2) &\quad \text{with} \\
  \#\text{leaves} &= n \quad \text{and} \\
  \Theta(n) &\quad \text{with}
\end{align*}
\]

Conclusions
- \( \Theta(n \log n) \) grows more slowly than \( \Theta(n^2) \).
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for \( n > 30 \) or so.
- Go test it out for yourself!
More examples – iterative method (1)

NoNeed(n){
    • If (n<1) return ;
    • Print(*)
    • NoNeed(n-1)
}

Recursion formula: \( T(n) = c + T(n-1) \), where \( T(1) = c \). We can solve it using the iteration method:

\[
T(n) = c + T(n-1) \\
=c + (c + T(n-2)) = 2c + T(n-3) = \ldots = (\text{pick } k<n) \\
kc + T(n-k) = \text{ (setting } k = n-1) \ldots \\
(n-1)c + T(1) = nc \]

Example 2

NoNeed(n){
    if (n<1) return ;
    for( i=1 ; i<n ; i++)   print(*)
    NoNeed(n-1)
}

Recursion formula: \( T(n) = cn + T(n-1) \), where \( T(1) = c \). We can solve it using the iteration method:

\[
T(n) = cn + T(n-1) \\
=cn + [cn+(n-1) + T(n-2)] \\
=cn + [cn+(n-1) + \ldots + T(n-k)] \\
=cn + [cn+(n-1) + \ldots + [cn+(n-k)] + T(1)] \\
=cn + 2n + 3 + \ldots + n + 1 = \Theta(n^2).
\]
Examples 3

We know that each iteration takes $O(1)$ times. Need to find the number of iterations.

- 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 $j$ th iteration $k=2^j$

• Assume $j$ iterations occurs until the loop exits. After the last one we have that $k=2^j < 2n$.

• Taking $\log_2$ from both sides, we have that
  \[
  \log_2(k) = \log_2(2^j) < \log_2(2n) \quad \text{or} \quad \log_2(n) + j \leq \log_2(n) + \log_2(n) = \log_2(n^2) = \log_2(n).
  \]

• Homework: Prove $T(n) = \Theta(\log n)$

Examples 4

Recall: $\log(ab) = \log(a) + \log(b)$
\[
\log(a^b) = b \log a
\]
\[
\log_a(x) = \frac{\log(x)}{\log_a}
\]

Read(n);
while(k $\leq$ n) k=2k;

• 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 $T(n)=O(n^2)$.

Is it true that the running time is $\Omega(n^2)$?
That is, is the running time $\Theta(n^2)$?

Examples 4

read(n);
for(i=1; i $\leq$ n; i++)
  for(j=i; j $\leq$ n; j += i)
    print("*");

• More "sensitive" analysis:
  • For $i=1$ we run through $j=1,2,3,\ldots,n$, total $n$ times.
  • For $i=2$ we run through $j=2,4,6,8,10\ldots,n$, total $n/2$ times.
  • For $i=3$ we run through $j=3,6,9,12\ldots,n$, total $n/3$ times.
  • For $i=4$ we run through $j=4,8,12,16\ldots,n$, total $n/4$ times.
  • For $i=n$ we run through $j=n$, or...
  • Summing up: $T(n)=n+n/2+n/3+n/4+\ldots+n/n=\sum_{i=1}^{n} \frac{n}{i} = n \ln n$

Reminder: Geometric sum

• Let $a$ be a constant
• Recall: $1+a+a^2+\ldots+a^t=\frac{(1-a^{t+1})}{(1-a)}$.

• If $a<1$ then $1+a+a^2+\ldots+a^t+\ldots$ (infinite sum) 
  \[
  = \frac{1}{1-a}
  \]
Example 5

```plaintext
read(n); a = 0.7;
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 \( a^kn \) times.

• Let \( t \) be the number of iterations of the outer loop. Then the total time
  \[ t = n + an + a^2n + a^3n + \ldots = \frac{n}{1-a} = O(n). \]

  • Same analysis holds for any \( a < 1 \)

Properties of big-\( \Omega \)

- Claim: if \( T_1(n) = \Omega(g_1(n)) \) and \( T_2(n) = \Omega(g_2(n)) \) then
  \[ T_1(n) + T_2(n) = \Omega(g_1(n) + g_2(n)) \]

- Example: \( T_1(n) = \Omega(n^2), T_2(n) = \Omega(n \log n) \) then
  \[ T_1(n) + T_2(n) = \Omega(n^2 + n \log n) = \Omega(n^2) \]

- Proof: We know that there are constants \( n_1, n_2, c_1, c_2 \) s.t.
  • for every \( n > n_1 \), \( T_1(n) \leq c_1 g_1(n) \) (definition of big-\( \Omega \))
  • for every \( n > n_2 \), \( T_2(n) \leq c_2 g_2(n) \) (definition of big-\( \Omega \))

  • Now set \( n' = \max\{n_1, n_2\}\), and \( c = c_1 + c_2 \) then
    • for every \( n > n' \) we have that
      \[ T_1(n) + T_2(n) \leq c_1 g_1(n) + c_2 g_2(n) \leq c' g_1(n) + c' g_2(n) = c' (g_1(n) + g_2(n)) \]

More properties of big-\( \Omega \)

- Claim: if \( T_1(n) = O(g_1(n)) \) and \( T_2(n) = O(g_2(n)) \) then
  \( T_1(n) \cdot T_2(n) = O(g_1(n) \cdot g_2(n)) \)

- Example: \( T_1(n) = O(n^2), T_2(n) = O(n \log n) \) then
  \( T_1(n) \cdot T_2(n) = O(n^2 \log n) \)

  • Similar properties hold for \( \Theta \)

The lower bound trick -

Example one - on the board
The lower bound trick – Second example

We are about to insert \( n \) keys \( \{k_1, \ldots, k_n\} \) into an empty AVL tree. How much time would it take?

Upper bound: When the \( i+1 \) key is inserted, the tree contains \( i \) keys, so its height is \( O(\log i) \), and an insert operation takes \( O(\log i) \) which is also \( O(\log n) \).

So the overall running time is:

\[
O(\log 1) + O(\log 2) + O(\log 3) + O(\log 4) + \cdots + O(\log n) \\
= O(\log n) + O(\log n) + O(\log n) + O(\log n) + \cdots + O(\log n) = O(n \log n)
\]

This is an upper bound. What is the lower bound?

- \( \Omega(n) \)?
- \( \Omega(n+1) \)?
- \( \Omega(2^n) \)?
- \( \Omega(n \log n) \)

The lower bound trick – a less trivial example

We demonstrate this trick by giving an \( \Omega(n \log n) \) bound on the time \( T(n) \) required to insert \( n \) keys into an (initially empty) balanced search tree.

The \( i \)th insertion takes \( K \log(i) \) time (for a constant \( K \), that we ignore). Hence:

\[
\sum_{i=1}^{n} \log i = \\
\log 1 + \log 2 + \cdots + \log(\frac{n}{2} - 1) + \log(\frac{n}{2}) + \log(\frac{n}{2} + 1) + \cdots + \log n \\
\geq \log(\frac{n}{2}) + \log(\frac{n}{2} + 1) + \log(\frac{n}{2} + 2) + \cdots + \log n \\
\geq \log(\frac{n}{2}) + \log(n) = \log(n) = \Omega(n \log n)
\]