CS 240: Data Structures and Data Management

Chris Thomson

Winter 2013, University of Waterloo

Notes written from Alejandro López-Ortiz's lectures.

1 Introduction & Code Optimization

1.1 Course Structure

The grading scheme is 50% final, 25% midterm, and 25% assignments. There are five assignments, usually due on Wednesday mornings at 9:30 am. There are several textbooks for the course, all of which are optional but recommended. The textbooks cover roughly 80% of the course content. There are also some course notes published online from previous terms, however the lectures will not necessarily follow those notes strictly.

See the course syllabus for more information.

1.2 CS as the Science of Information

So far in our undergraduate careers, computer science has meant programming. However, programming is only a subset of computer science. Computer science is the **science of information**.

What do we want to do with information? We want to:

- **Process it**. Programs ≡ algorithms.

- **Store it**. We want to encode it. This also leads to information theory. Storing information involves data structures, databases, (file) systems, etc., all of which are searchable in some way.

- **Transmit it**. We want to transmit information over networks. This process involves coding theory.

- **Search it**. First, we have to structure it with data structures and/or SQL databases. Information retrieval is the process of searching for textual information instead of numerical information.

- **Mine it**. This involves artificial intelligence and machine learning.

- **Display it**. Information needs to be displayed using computer graphics (CG) and user interfaces (UI, partially related to psychology).

- **Secure it**. Encryption and cryptography are important. Information should also be stored redundantly to prevent harm from catastrophic events.

1.3 Objectives of the Course

- **Study efficient methods of storing, accessing, and performing operations on <u>large</u> collections of data.**

  "Large" is subjective – your data needs to be large enough to justify the additional mental complexity.

  Typical operations:

  – Insert new item.

- "Deleting" data (flagging data as "deleted", not actually deleting the data).
- Searching for data.
- Sorting the data.

Examples of "large data":

- The web.
- Facebook data.
- DNA data.
- LHC (Large Hadron Collider) measurements (terabytes per day).
- All the music from iTunes, for finding duplicate songs, etc.

- **There is a strong emphasis on mathematical analysis.**

  The performance of algorithms will be analyzed using order notation.

- **The course will involve abstract data types (objects) and data structures.**

  We will see examples that have to do with:

  - Databases.
  - Geographic databases.
  - Text searching.
  - Text compression.
  - Web searching.

1.4   Code Optimization & Measuring Performance

Richard Feynman was in charge of the computer group on the Manhattan Project. He made his code run 10x faster.

The initial method to measure performance was to use a **wall clock**. Initial studies looked like this:

Data size	A	B
3	1	3
5	2	9
10	4	10
20	16	11

However, computers were getting better, so we couldn't use wall clocks anymore. Results were not robust enough due to the qucik progression of performance increases in terms of computing power. Moreover, because of the differences in architecture, the same program may have a different execution time on two different computer models.

Idea: rather than comparing algorithms using seconds, we should compare algorithms using the number of operations required for the algorithm to run.
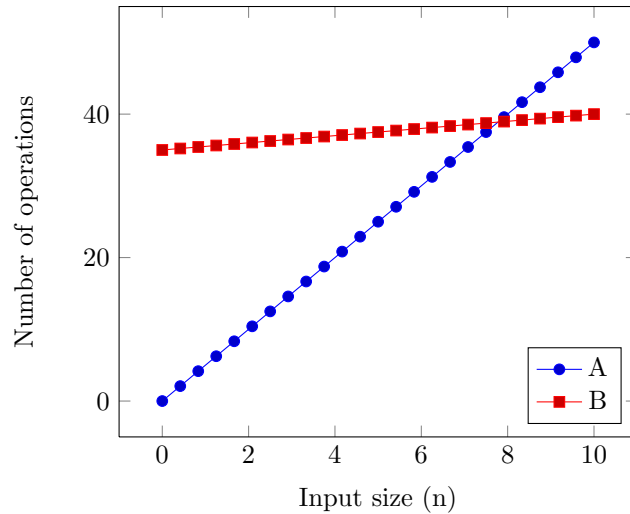
Figure 1: A comparison of two algorithms A and B.

- Express algorithm using pseudocode.

- Count the number of primitive operations.

- Plot the number of operations vs. the input size.

Note that A and B are plotted as continuous functions, however they are not actually continuous – we join all of the points for readability purposes only.

In the long run, you may want to use algorithm B even if algorithm A is better in some cases, because the benefits of A will be short lived as n grows.

Hence, comparing the programs has been transformed into comparing functions. We use order notation to measure the long-term growth of functions, allowing us to choose the smaller function, which corresponds to the faster algorithm.

2   Order Notation

The time for algorithm A on input of size n is:

$$\underbrace{2n \log n}_{\text{sort}} + \underbrace{1.5n^3}_{\text{mult}} + \underbrace{22n^2 - 3n}_{\text{additions}} + \underbrace{7}_{\text{setup}}$$

On a different machine, the same algorithm A may be $2n \log n + 9n^3 + 10n^2 - 3n + 7$. These give a **false sense of precision**. These specifics can also be quite costly to determine. In the 1960s, Don Knuth proposed **order notation** as a way to analyze the general quality of algorithms in an accurate, cost- and time-efficient way by ignoring precise runtimes.

4

2.1 Formal Definitions

Order notation represents algorithms as functions. We say a function $f(n)$ relates to a certain order $g(n)$ using different notation depending on which relation we're interested in.

Relation	Functions
$3 \leq 7$	$f(n) = O(g(n))$
$8 \geq 7$	$f(n) = \Omega(g(n))$
$7 = 7$	$f(n) = \Theta(g(n))$
$3 < 7$	$f(n) = o(g(n))$
$7 > 3$	$f(n) = \omega(g(n))$

Here's an easy way to remember the correspondence between a relation and its order notation symbol: the greedy operators, $\leq$ and $\geq$, are uppercase letters O and Ω. The less greedy operators, $<$ and $>$, are the same letters but in lowercase (o and ω).

Order notation only cares about the long run. An algorithm can violate the relation early – we're interested in its asymptotic behavior.

Definition ($\leq$). $f(n) = O(g(n))$ if there exist constants $c > 0, n_0 > 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Definition ($\geq$). $f(n) = \Omega(g(n))$ if there exist constants $c > 0, n_0 > 0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

Definition ($=$). $f(n) = \Theta(g(n))$ if there exists constants $c_1, c_2 > 0$ such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

The equality relation is effectively sandwiching the algorithm $f(n)$ between two other functions (the squeeze theorem). If it's possible to sandwich the algorithm between two multiples of the same $g(n)$, then $f(x)$ is said to be equal to the order $g(n)$.

Definition ($<$). $f(n) = o(g(n))$ if for all $c > 0$ there exists constant $n_0 > 0$ such that $f(n) < c \cdot g(n)$ for all $n \geq n_0$.

Definition ($>$). $f(n) = \omega(g(n))$ if for all $c > 0$ there exists constant $n_0 > 0$ such that $f(n) > c \cdot g(n)$ for all $n \geq n_0$.

2.2 Order Notation Examples

Example 2.1. $2n^2 + 3n + 11 = O(n^2)$

We need to show that there exists constants $c > 0$ and $n_0 > 0$ such that $2n^2 + 3n + 11 \leq cn^2$ for all $n \geq n_0$.

Let $c = 3$. Simplifying gets us $3n + 11 \leq n^2$. This holds for $n_0 = 1000$, for instance.

Example 2.2. $2n^2 + 3n + 111 = \Theta(n^2)$

In order to show equality (Θ), we need to show both the $\leq$ and $\geq$ cases. In the previous example, the $\leq$ case was shown. For the $\geq$ case, we need to show $n^2 \leq c \cdot (2n^2 + 3n + 11)$.

Let $c = 1$. Simplifying gets us $n^2 \leq 2n^2 + 3n + 11$, which gives us $0 \leq n^2 + 3n + 11$. This holds for $n_0 = 0$.

Example 2.3. $2010n^2 + 1388 = o(n^3)$

We must show that for all $c > 0$, there exists $n_0 > 0$ such that $2010n^2 + 1388 \leq c \cdot n^3$ for all $n \geq n_0$.

$$\frac{2010n^2 + 1388}{n^3} \overset{?}{\leq} \frac{c \cdot n^3}{n^3} \implies \frac{2010}{n} + \frac{1388}{n^3} \overset{?}{\leq} c$$

There is a **trick to prove f(n) = o(g(n))**: show that $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$.

2.3   A note about the use of $=$

$$\underbrace{2n^2}_{\text{specific function}} \in \underbrace{O(n^2))}_{\text{set of functions}}$$

The use of the equality operator ($=$) in order notation is not the same as in most other areas of mathematics, since a single element of a set cannot strictly be equal to the set itself (even $a \neq \{a\}$). The $=$ in order notation is not a true equality – it's just notation. The $\in$ operator is semantically more correct.

2.4   Performance of Algorithms

We have made some progression in how we determine the performance of algorithms.

- Comparing single runs.
- Comparing measured curves (multiple runs).
- Produce analytical expressions for the curves.
- Simplify using order notation.

Let's say we have the problem of integrating a mathematical function. Our program numerically integrates the function, and we're trying to analyze the algorithm behind that process. The timing would vary depending on the mathematical function and preciseness required.

If we conduct multiple runs of our algorithm, we'll get different times. Plotting all of the results (time vs. input size) will result in data that is not a function, because the data fails the vertical line test, since for a given input size we have multiple times.

We need to decide on a convention for these cases where we have multiple time values. In this course, we'll usually look at the worst case, however in some situations examining the best or average cases could be useful.

Finding the average case is hard. To produce an appropriate average, we need an input distribution.

3   Formalism

Definition. A **problem** is a collection of questions and (correct) answer pairs.

Example 3.1. Informally, the problem is "multiplying two numbers." More formally, the problem could be represented as: (2 x 3, 6), (3 x 4, 12), (1 x 5, 5), etc.

Definition. An **instance** of a problem is a specific question and answer pair, (Q, A).

Definition. The **input** of a problem is the question. Example: 3 x 4.

Definition. The **output** is the only correct answer. Note: there is only one correct answer under this definition. "Multiple" correct answers can always be reduced to some canonical form that represents the one correct answer.

Definition. An **algorithm** is a mechanical method to produce the answer to a given question of a problem.

Definition. The **size of the input** is the number of bits, characters, or elements in the question. This definition will vary depending on what's most appropriate for the given question.

Long division in elementary school was the first time a problem's complexity was directly related to the input size. That's not always the case, however.

Definition. We say an algorithm **solves** a problem if for every question Q it produces the correct answer, A.

Definition. A **program** is an implementation of an algorithm using a specified computer language.

Example 3.2. We want to sort n numbers. One instance of this problem Q is:

$$(\underbrace{(5,1,3,2,4)}_{Q}, \underbrace{(1,2,3,4,5)}_{A})$$

For a problem in this form, we'll say the size of Q is $|I| = 5$. Why? Counting the number of elements is the most logical definition in this case.

This course emphasizes algorithms rather than programs. We're computer scientists, so we care about the algorithm and the speed/efficiency of that algorithm.

A problem Π can have several correct algorithms that solve it. Our goal is to find efficient solutions to Π. How do we do that?

1. **Algorithm design**. Find a solution(s) to Π.

2. **Algorithm analysis**. Assess the correctness and efficiency of the algorithm.

In this course, we're mostly interested in algorithm analysis. Algorithm design will be covered in more detail in CS 341.

3.1 Timing Functions

A timing function is a function T_A such that:

$$T_A : \{Q\} \to \mathbb{R}^+$$

where $\mathbb{R}^+$ is the set of positive real numbers.

$T_A(Q)$ is the time taken by algorithm $\mathcal{A}$ to compute the answer to Q.

We also want to define a general timing function for a particular problem, regardless of algorithm:

$$T(n) = max\{T_A(Q)\}$$
$$T_{avg}(n) = avg\{T_A(Q)\}$$
$$T_{min}(n) = min\{T_A(Q)\}$$
$$\vdots$$

If we had two solutions (algorithms) $T_A(n)$ and $T_B(n)$, we could use order notation to compare the quality of A and B.

Note that some problem instances are easier than others to solve, even with the same input size. Most people find it easier to multiply 0×7 than 8×7, for instance, despite those two problems having the same input size.

4 Analysis of Algorithms

In order to analyze an algorithm, we count the number of <u>basic</u> operations that the algorithm performs.

Example 4.1. Let's say our algorithm is to compute $x^2 + 4x$ and assign it to a variable called `result`. That involves seven basic operations:

1. Read x from memory.

2. Compute $x \cdot x$.

3. Assign the result of $x \cdot x$ to a variable, `pr1`.

4. Compute $4 \cdot x$.

5. Assign the result of $4 \cdot x$ to a variable, `pr2`.

6. Compute `pr1` + `pr2`.

7. Assign the result of `pr1` + `pr2` to a variable, `result`.

The number of operations remains the same across machines, but the actual running time of an algorithm will differ from machine to machine (which one of the reasons why we use order notation).

We only count the number of <u>basic</u> operations. There are various definitions of what a "basic" operation actually is (especially with regards to variable assignments), but these are some common operations that are considered to be basic:

- Add numbers of reasonable size (numbers that can be added primitively).

- Multiply numbers of reasonable size (numbers that can multiplied primitively).

- Access an index in an array.

- Store a value in memory (variable assignments).

- Read a value from memory.

Be careful. Some languages disguise complexity well. Just because the syntax is simple doesn't necessarily mean it's a basic operation, and doesn't guarantee that the operation runs in constant time.

4.1 Techniques for Algorithm Analysis

- Straight line programs (no loops). Simply tally up the basic operation count.

- Loops (`for`/`while`). You need to add the number of operations for each pass on the body of the loop.

Example 4.2. Analyze the following algorithm.

for $i = a$ to b **do**
| < straight line program > T_L
end

This program will run with $\sum_{i=a}^{b} T_L(i)$ operations.

Whenever you have a loop in your algorithm, you should expect to get a summation in your timing function.

Example 4.3. Analyze the following algorithm.

for $x = 0$ to 10 **do**
| pr1 = x * x;
| pr2 = 4 * x;
| result = pr1 + pr2;
| print result;
end

This program will run with $\sum_{i=0}^{10} 4 = 44$ operations (depending on your definition of "basic" operations).

Operations like addition and multiplication are primitive for numbers of a reasonable size. Once numbers get large enough to exceed certain limits, we have to resort to some trickery to perform those operations, which adds additional complexity, making them no longer "basic" operations.

We can be a bit sloppy and use the worst case, then say the actual algorithm is $\leq$ what we calculated.

Example 4.4. Analyze the following algorithm. Test 1 (n).

```
sum = 0;
for i = 1 to n do
 |  sum = sum + i
end
return sum
```

This program will run with $1 + (\sum_{i=1}^{n} 1) + 1 = 2 + n = \Theta(n)$ operations.

Example 4.5. Analyze the following algorithm.

```
sum = 0;
for i = 1 to n do
    sum = sum + (i − j)²;
    sum = sum²;
end
return sum
```

This program will run with time:

$$1 + \sum_{i=1}^{n} \left[\sum_{j=1}^{i} 4 \right] + 1$$

$$= 2 + \sum_{i=1}^{n} 4i$$

$$= 2 + 4 \sum_{i=1}^{n} i$$

$$= 2 + 4 \cdot \frac{n(n+1)}{2} \quad \text{(Gauss)}$$

$$= 2 + 2n^2 + 2n$$

$$= \Theta(n^2)$$

We can be pessimistic, assume worst-case scenario, and say that it runs with time:

$$2 + \sum_{i=1}^{n} n$$

$$= O(n^2)$$

Note that $O(n^2)$ is a less precise result than $\Theta(n^2)$, but in some cases that's good enough.

Order notation helps make algorithm analysis easier by allowing us to throw away any specific constants, as long as the order remains untouched. The entire analysis process happens within the context of order notation, so you can just start dropping constants immediately.

Keep in mind, it is possible to create a bad over-estimation. You have to be smart about it.

Example 4.6. Analyze the following algorithm. Test 2 (A, n).

```
max = 0;
for i = 1 to n do
    for j = 1 to n do
        sum = 0;
        for k = i to j do
            sum = A[k] + sum;
            if sum > max then
                max = sum;
            end
        end
    end
end
return max
```

This is the **maximum subsequence problem**. The input is an array of integers $A[1 \ldots n]$, and the output is the consecutive run with the largest sum.

Sample sequence:

2	-4	1	3	-2	8	-1

The running time of this program is $max\left\{ \sum_{i=1}^{j} A[k] \big| 1 \leq i \leq j \leq n \right\}$.

Example 4.7. Analyze the following algorithm.

```
for i = 1 to n do
    for j = 1 to i do L_1[i]
        for k = 1 to j do L_2[i, j]
            x = x + 1
        end
    end
end
```

11

$$\sum_{i=1}^{n} L_1(i) = \sum_{i=1}^{n}\left[\sum_{j=1}^{i} L_2(i,j)\right]$$

$$= \sum_{i=1}^{n}\left[\sum_{j=1}^{i}\left[\sum_{k=1}^{j}\Theta(1)\right]\right]$$

$$= \sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=1}^{j} 1$$

$$= \sum_{i=1}^{n}\sum_{j=1}^{i} j$$

$$= \sum_{i=1}^{n}\frac{i(i+1)}{2}$$

$$= \frac{1}{2}\sum_{i=1}^{n}(i^2 + i)$$

$$= \frac{1}{2}\left[\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i\right]$$

$$= \frac{1}{2}\left[\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2}\right]$$

$$= \Theta(n^3)$$

Alternatively, we could use a lazier process to determine that this algorithm is $O(n^3)$, wihch is less precise than saying the algorithm is $\Theta(n^3)$. The lazy way is to say each of the nested loops will run in $\leq n$ operations in the worst case, which can be multiplied out to n^3.

4.2 Math Review

4.2.1 Exponentiation

Exponents have a number of important properties, including:

$$b^0 = 1$$
$$b^1 = b$$
$$b^{1/2} = b^{0.5} = \sqrt{b}$$
$$b^{-1} = \frac{1}{b}$$
$$b^a \cdot b^c = b^{a+c}$$
$$(b^a)^c = b^{ac}$$

4.2.2 Logarithms

Definition. $\log_b a = c$ if and only if $a = b^c$. If $b = 2$, we write $\lg a$.

There are a number of log identities you should be aware of:

$$\log_b(a \cdot c) = \log_b a + \log_b c \tag{1}$$

$$\log\left(\frac{a}{c}\right) = \log_b a - \log_b c \tag{2}$$

$$\log(a^c) = c \cdot \log_b a \tag{3}$$

$$b^{\log_c a} = a^{\log_c b} \tag{4}$$

$$\log_b a = \frac{\log_c a}{\log_c b} \tag{5}$$

Identity (1) is useful because it allows you to go from multiplication to addition. You can avoid nasty multiplication operations by using this identity.

Identity (4) is the prof's favorite. If b and c are constants, $\log_b n = \Theta(\log_c n)$ – that is, we don't care about the base of the logarithm in the context of order notation.

4.2.3   Recursive Definitions

Definition. The **factorial** of a number n is represented by $n!$. Informally, $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$. More formally:

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

Definition. The **fibonacci numbers** are defined by:

$$F_i = \begin{cases} 0 & i = 0 \\ 1 & i = 1 \\ F_{i-2} + F_{i-1} & i > 1 \end{cases}$$

4.2.4   Summations

There are a few common summations you should be aware of:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^k \approx \frac{n^{k+1}}{k+1} = \Theta(n^{k+1})$$

You should also be familiar with the **geometric series**:

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \text{ where } a < 1$$

4.3 Useful Properties of Order Notation

1. $f(n) = \Theta(a \cdot f(n))$ for $a > 0$.

2. Transitivity. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

3. $[f(n) + g(n)] = \Theta(\max\{f(n), g(n)\})$

4. $a_0 + a_1 x^1 + a_2 x^2 + \ldots + a_n x^n = \Theta(x^n)$, where a_i are constants, $x > 1$, and $a_n > 0$.
   This is just a common case of (3).

5. $n^k = O(a^n)$ for $a > 1$.

6. $\log^k n = o(n^b)$ for $k > 0$ and $b > 0$ (where $b \in \mathbb{R}$, not necessarily the base of the logarithm).

You can use these properties in proofs *unless* you're requested to write a proof from first principles.

4.4 Orders We Aim For

When we are designing an algorithm, we aim for:

1. Constant time: $\Theta(1)$

2. Logarithmic time: $\Theta(\lg n)$

3. Poly log: $\Theta((\lg n)^k)$

4. Linear complexity: $\Theta(n)$

5. $n \log n$ complexity: $\Theta(n \lg n)$

6. Quadratic: $\Theta(n^2)$

7. Cubic: $\Theta(n^3)$

8. Polynomial: $\Theta(n^k)$

9. Exponential: $\Theta(b^n)$

4.5   Maximum Subsequence Problem

Recall the maximum subsequence problem from earlier:

```
max = −∞;
for i = 1 to n do
    for j = 1 to n do
        total = 0;
        for k = i to j do
            total = total + A[k];
            if total > max then
                max = total;
            end
        end
    end
end
return max
```

Also, recall our sample sequence A:

2	-4	1	3	-2	8	-1

Note that the longest run (the subsequence with the largest sum) in this sample sequence is from 1 to 8, which is a subsequence that sums to 10.

The maximum subsequence problem has many applications. It is used for DNA matching, for instance. There is a score for accuracy / closeness of the match. You get a score for various runs within a DNA sequence. If there is a long enough run(s) then you have probably identified the correct person. Google may also use the maximum subsequence problem as part of its algorithm for determining typos or other terms to match a particular search query (such as the word's plural form).

Our naïve solution is not good if n is sufficiently large, such as $10,000$ or $1,000,000,000,000$. Notice that the innermost loop recalculates the sum unnecessarily every time j is incremented. You really just need to add the new element to the previous result. That gives us this more efficient algorithm:

```
max = −∞;
for i = 1 to n do
    total = 0;
    for j = i to n do
        total = total + A[j];
        if total > max then
            max = total;
        end
    end
end
return max
```

This algorithm runs in $O(n^2)$ time. We could do better, but that's okay for now. Our code is now 10,000 times faster on an input of size 10,000.

5 Recursive Analysis

We are going to analyze **mergesort**. Recall: mergesort involves sorting the two halves separately, then merging the sorted items into a single list again. We merge items $(1 \ldots n/2)$ and $(n/2 + 1 \ldots n)$ separately, then merge the two lists.

The number of operations mergesort requires is:

$$T_{ms}(n) = T_{ms}\left(\frac{n}{2}\right) + T_{ms}\left(\frac{n}{2}\right) + n$$
$$= 2T_{ms}\left(\frac{n}{2}\right) + n$$

$$T_{ms}(1) = 1 \text{ (base case)}$$

The method you use here is **guess and check**. For example, for $n = 4$, we have:

$$T_{ms}(4) = 2T_{ms}(2) + 4$$
$$= 2(2T_{ms}(1) + 2) + 4$$
$$= 4T_{ms}(1) + 4 + 4$$
$$= 4 + 4 + 4$$

$$T_{ms}(8) = 8 + 8 + 8 + 8$$

$$T_{ms}(n) = n \cdot \lg n$$

You can check this with induction. That is, assuming this holds for $n = 2$, show that it holds for any n. If $T_{ms}(n) = n \lg n$, then:

$$T_{ms}(n) = 2T_{ms}\left(\frac{n}{2}\right) + n$$
$$= 2 \cdot \frac{n}{2} \cdot \lg \frac{n}{2} + n$$
$$= n \cdot \lg \frac{n}{2} + n$$
$$= n \cdot (\lg n - \log 2) + n \text{ by log identity}$$
$$= n(\lg n - 1) + n$$
$$= n \lg n - n + n$$
$$= n \lg n$$

Therefore, mergesort has time $\Theta(n \lg n)$.

6 Abstract Data Types

6.1 Stack

A **stack** is a collection of items, which supports the operations `push` (insert an item), `pop` (remove most recently inserted item), `peek` (look at the last item), and `isEmpty`. `push` and `pop` are the most commonly supported operations.

6.1.1 Stack Implementations

Common ways to implement a stack are:

- **Linked List**. You maintain a pointer to the top of the stack. When you `push` or `pop`, you change your pointer to the next or previously item as appropriate. This method is more flexible, but pointers can be a pain.

- **Array**. You keep track of where the last item is in the array. You may need to re-size your array, but at least you don't have to worry about pointers.

6.2 Queue

A **queue** is a data structure where you `insert` at the end and remove (`dequeue`) from the front, just like the way a queue (a lineup) works in real life.

Implementations are similar to implementing a stack. You maintain pointers to the start and end of the array or linked list.

6.3 Priority Queue

A priority queue is a collection of elements similar to a queue, except each element has a priority assigned to it and elements are `pop`ped in order of priority (not in order of arrival).

A priority queue supports these operations:

- `insert(x, p)` – inserts item x with priority p.

- `deleteMin()` – deletes. Used when the queue is defined to be such that lower numbers indicate higher priority. This deletes the element with the lowest p.

- `deleteMax()` – deletes. Used when the queue is defined to be such that higher numbers indicate higher priority. This deletes the element with the highest p.

- `peek()` – view the top element without deleting it.

6.3.1 Applications of Priority Queues

- To-do lists in real life.

- To-do lists in the operating system (used for multitasking; the processor can only really do one thing at a time, so it does a bit of each task as determined by priority).

- Sorting. Insert the elements from an array A and then when you delete them you'll get them back in sorted order.

Priority queues can be used for sorting purposes. It allows us to sort without worrying about the underlying sorting implementation. However, if we want to know how good of a sorting algorithm it is, we would need to examine the implementation of `insert` and `deleteMin`.

Here's some pseudocode for sorting with a priority queue:

```
for i = 0 to n - 1 do
|   PQ.insert(A[i], A[i]); // key and priority are the same
end
for i = 0 to n - 1 do
|   A[i] = PQ.deleteMin();
end
```

6.3.2 Implementations of Priority Queues

1. Use **unsorted arrays**.

   - `insert(x, p)` takes O(1) time because you simply place the element at the end of the array.
   - `deleteMin()` takes O(n) time since you need to walk the array (keeping track of the current min you've seen so far), then replace the deleted element with the last element of the array. O(n) + O(1) = O(n) time.

   Sorting with unsorted arrays takes $O(n^2)$ time, since for each element you're deleting you need to walk the entire array. Clearly, this is not a good sorting algorithm.

2. Use **heaps**. A heap is a <u>complete</u> binary tree that has the <u>heap property</u>.

   Recall: a **complete (or perfect) binary tree** is defined as a binary tree such that:

   - Every node has 0 or 2 children, except the rightmost leaf in the bottom level, which may have one only child.
   - All the leafs appear consecutively in the bottom two levels.
   - Deeper leaves are leftmost in the tree.

   You can think of a complete binary tree as trying to fill the tree from left-to-right, going deeper and deeper as necessary.
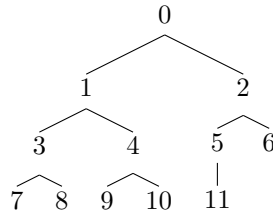

Figure 2:   A complete binary tree.

   The **heap property** is the idea that the priority of a node is higher than that of all of its children (if any). In a min-PQ, this means the p value of all children is a larger number than that of the parent. In a max-PQ, the childrens' p values must all be smaller than that of their parent.

```
            5
          /   \
       10       7
      /  \     /  \
    12    14  13    8
```
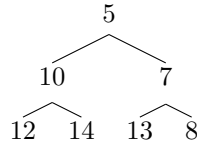
Figure 3: A min-PQ tree that satisfies the heap property.

You can have multiple elements with the same priority. You settle those disputes arbitrarily.

This is <u>not</u> a binary search tree, so the larger element does not need to be on the right. There is no order among siblings, as seen in Figure 3. There is also no relationship with the parent's siblings – it's only a relationship with the parent.

Next, we'll look into **filling holes**. If you delete an element from the tree, you delete the element at the top, but then you have a hole that needs to be filled. But how? We take the last element in the tree (the rightmost element at the bottom level) and swap it into the top position, then we get it to **boogie down**.

The process of **boogieing down** is where an element (in this case, the top element) looks at its children and swaps with a child that has lower priority than itself (if any). It does this recursively. It continues to recurse until the child is of higher priority than its parent, or until you reach a leaf node. Swap places with the child with the highest priority.

Insertion uses the process of **bubbling up**. We place the new element in the first available spot, which will be in the bottom level. The new element exchanges with their parent if the new element has higher priority than the parent, and that continues as far as needed, until it doesn't swap or until it reaches the top.

Implementing a heap as an array (also known as a **l33t hacking trick**):

You can actually implement a heap without pointers, using an array instead. The array indices follow the pattern as indicated in Figure 2 by inserting at the end of the array.

The parent of a node at index m is $\lfloor \frac{m-1}{2} \rfloor$. The children for a node at index m are located at index $2 \cdot m + 1$ and $2 \cdot m + 2$.

To perform a `deleteMin`, we erase element 0 and replace it with the last element in the array. We then perform a boogie down on element 0. Before computing children, we must perform boundary checking to ensure there actually are children (that is, we haven't run off the end of the array).

Here's the pseudocode for the bubble up process:

```
while ⌊v−1/2⌋ ≥ 0 and A[⌊v−1/2⌋].p > A[v].p do   [parent exists and has higher priority]
    swap(v, ⌊v−1/2⌋);
    v = ⌊v−1/2⌋;
end
```

Here's the pseudocode for the boogie down process:

```
while 2v + 1 ≤ last do   [v has at least one child]
    u = argmin{A[2v + 1], A[2v + 2]}; // argmin returns the array index of the min
    if A[u].p < A[v].p then
        swap(u, v);
        v = u;
    else
        break;
    end
end
```

When implementing boogieing down, you'll have to handle boundary cases where there is only one child, etc., which were not handled in this pseudocode.

7 Sorting

7.1 PQ Sort

We can sort data using a priority queue.

Both `insert` and `deleteMin` take O(lg n) time where n is the number of elements in the tree. Bubble up and boogie down each take O(h) = O(lg n) time, where h is the height of the tree.

Note that the number of elements in a tree is between 2^{h-1} and 2^h. PQ sort takes O(n lg n) time, which is the same as mergesort.

7.2 Heap Sort

Heap sort is a specialization of PQ sort. We can actually insert things even faster into the heap, though.

Claim: if all the elements to be inserted are given at once we can build the heap in $\Theta(n)$ time, which is faster than $\Theta(n \lg n)$.

Proof. We're going to show that we can run this algorithm in $\Theta(n)$ time.

```
Heapify(A); // A is an array
n = size(A) - 1;
for i = ⌊n−1/2⌋ down to 0 do
    boogie_down(A, i);
end
```

The number of comparisons for boogie down is:

$$\frac{n}{2} + 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \ldots + 1 \cdot \lg n$$

$$\sum_{i=1}^{\lg n} \frac{n}{2^i} \cdot i$$

The cost of boogieing down n elements is:

$$T(n) = T\left(\frac{n}{2}\right) + \frac{n}{4} = T\left(\frac{n}{4}\right) + \frac{n}{8} + \frac{n}{4}$$

This is the case because there are $\frac{n}{4}$ elements that can go one level further down. So we have:

$$\begin{aligned}
T\left(\frac{n}{2}\right) &= T\left(\frac{\frac{n}{2}}{2}\right) + \frac{\frac{n}{2}}{4} \\
&= T\left(\frac{n}{4}\right) + \frac{n}{8} \\
&= \frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \frac{n}{32} + \ldots \\
&= n\left(\frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \ldots\right)
\end{aligned}$$

This is true because $\sum_{i=0}^{\infty} \frac{1}{2^i} = 1 + \frac{1}{2} + \frac{1}{4} + \ldots = 2$ was proven by Zeno.

Aside: Zeno's proof of this involved a human and a turtle walking. The human walked at a pace of 1 metre per hour and the turtle moved at a pace of a $\frac{1}{2}$ metre per hour. The turtle was given a head start. The human would catch up with the turtle at point 2, but not any sooner because both the human and the turtle will have made progress on their race by the time the human reaches a point where the turtle was previously. This is known as **Zeno's paradox**. Unrelated: someone has called the dust that you can never sweep into a dust pan zenodust.

An alternate way to prove this is as follows:

$$\sum_{i=0}^{\lg n} \frac{n}{x^i} \cdot i = n \sum_{i=0}^{\lg n} \frac{1}{x^i} \cdot i$$

$$= n \sum_{i=0}^{\lg n} \tau^i \cdot i \text{ where } \tau = \frac{1}{x}$$

What we now have looks a bit like a derivative. Differentiating gives:

$$n \cdot \tau \sum_{i=0}^{\lg n} i \cdot 2\tau^{i-1}$$

We can integrate and then differentiate.

Therefore, we can heapify in $\Theta(n)$ time, so heap sort is $\Theta(n \lg n)$ and the constant will be lower than other methods, which is preferable. □

In practice, it's better to insert one by one, because our analysis is in the worst-case. When comparing the average case, inserting one by one is faster than heapifying. The worst case is really rare, so you should insert one by one (and pay more attention to the average case analysis in this situation).

A heap implemented using an array is called an **in-place implicit data structure**. You avoid pointers by using trickery in the organization of the data itself.

7.3   Quick Sort

Selection: suppose we want to find the minimum or maximum element in an unsorted array. The easy solution is to scan the array and report the answer in $\Theta(n)$ time. However, if you're asked to find the k-th largest element? The naïve solution would be to sort the entire array, which would take $\Theta(n \lg n)$ time.

We can do better. We can find the k-th largest element in an array in linear time in the worst case, using a method called quick select.

7.3.1   Quick Select

Given an array A, we want to find the k-th smallest (or k-th largest) element. For example, let's say we're looking for the 3rd smallest element and we're given this array A:

20	8	9	2	1	7	15	12

We now want to partition the array into two smaller sets of elements. We will first sort the elements onto the proper side of 20 (the first element):

8	9	2	1	7	15	12	20

Then, we'll continue on the side of 20 that happens to contain the element we're looking for. This time, we'll sort elements onto the proper side of 8 (which is the first element).

2	1	7	8	9	15	12	20

If we were to continue further, we would proceed on the left side of 8 because there are fewer (or equal) to 3 elements on that side.

We use a random element from the array called the **pivot** to partition the elements into those smaller than the pivot and those larger than the pivot. In general: | < p | p | > p |

Here's some pseudocode for quick select:

```
p = choose random pivot in A;
partition (A, p) obtaining position i of p;
if k < i then
│   quick_select(A[0, i - 1], k);
else if k > 1 then
│   quick_select(A[i + 1, ..., n - 1], k - i - 1);
else
│   return p;
end
```

In quick select, you only continue to recurse in the "half" that contains what we're looking for (unlike in quick sort where we continue to recurse on both sides).

In the worst case, this runs in $\Theta(n^2)$ time, since the random element chosen is always $\max(A)$, and we are searching for k = first element: | ... | p_3 | p_2 | p_1 |

Partitioning with two arrays is easy (i.e. creating a new array and inserting). Partitioning within the existing array is a bit more complicated, however that is a key advantage to quick select. Here's the pseudocode for partitioning within the same array:
A sample run of this algorithm is as follows:

```
swap(A[0], A[p]);
i = 1;
j = n - 1;
while true do
│   while i < n and A[i] < A[0] do
│   │   i = i + 1;
│   end
│   while j ≥ 1 and A[j] > A[0] do
│   │   j = j - 1;
│   end
│   if j < i then
│   │   break;
│   else
│   │   swap(A[i], A[j]);
│   end
end
```

← January 29, 2013

```
44 4 10 52 86 57 97 19 18 88 31 82 59 39
19 i -->>              44              <<-- j
19 4 10 18 86 57 97 44 52 88 31 82 59 39
           i -->>-<>-<<--   j
4 10 18 19 86 57 97 44 52 88 31 82 59 39 (final output)
```

In-place partitioning is one the reasons why quick sort (and quick select) is so good. Recall the selection problem, where we have to find a way to return the k-th element in the sorted order. Quick select is one solution to this problem.

Worst case: quick select runs in $T(n) = n + T(n-1) = n + (n-1) + (n-2) + \ldots + 1 = \Theta(n^2)$ time.

Average case: assume all permutations are equally likely. This is a probabilistic assumption on the input. The probability of choosing a "good" pivot is $\frac{1}{2}$. After r recursive calls, about half the pivots are good, which means the array was halved approximately $\frac{r}{2}$ times. After $4 \lg n$ recursive calls, we must be left with an array of size 1. Note that this is $4 \lg n$ and not $2 \lg n$ because we aren't dividing perfectly in half each time. We're also assuminng that randomness is somehow preserved on recursive calls.

Expected case: probabilities come from the algorithm itself. For example: select p at random in $A[0 \ldots n - 1]$, which preserves randomness. Regardless of the input distribution or partition algorithm, a pivot is "good" with probability $\approx \frac{1}{2}$. The expected case is that this algorithm runs in $T(n, k)$ time as follows:

$$T(n, k) = cn + \frac{1}{n}T(n-1, k) + \frac{1}{n}T(n-2, k) + \frac{1}{n}T(n-3, k) + \ldots + \frac{1}{n}T(n+1, k-1) + \frac{1}{n}T(n+2, k-2) + \ldots$$

$$= cn + \frac{1}{n}\left[\sum_{i=0}^{k-1} T(n-i-1, k-i-1) + \sum_{i=k+1}^{n-1} T(i, k) \right]$$

We could hand-wave and say:

$$T(n) \leq \frac{1}{2}T(n) + \frac{1}{2}T\left(\frac{3n}{4}\right) + cn$$

$$\leq 2cn + 2cn\left(\frac{3n}{4}\right) + 2c\left(\frac{9n}{16}\right) + \ldots + T(1)$$

$$\leq \frac{1}{2}\left[\frac{1}{2}T(n) + \frac{1}{2}T\left(\frac{3n}{4}\right) + cn \right] + \frac{1}{2}\left[\frac{1}{2}T\left(\frac{3n}{4}\right) + \frac{1}{2}T\left(\frac{9n}{16}\right) + c\frac{3n}{4} \right]$$

$$\leq \frac{1}{4}T(n) + \frac{1}{2}T\left(\frac{3n}{4}\right) + \frac{1}{2}cn + \frac{1}{4}T\left(\frac{9n}{16}\right) + c\frac{3n}{8} + cn$$

$$\leq \underbrace{T(1)}_{\text{constant}} + \underbrace{2cn \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i}_{\text{constant}}$$

$$= \Theta(n)$$

$\therefore$ We can quick select in linear time in the expected case.

We really don't need highly random bits for quick select – at least nowhere near as random as in cryptography. You generally need a good random source, however for this application (since security is not an issue), the current time may suffice as a random seed. If we can *easily* produce a sequence mechanically with a program, then it is not considered random. True randomness cannot be compressed.

7.3.2 Quick Sort Algorithm and Analysis

The general algorithm for quick sort is as follows.

if $n = 0$ **then return**;
p = choose random_pivot(0 ... n - 1);
i = partition(A, p);
QS(A[0 ... i - 1]);
QS(A[i + 1 ... n]);

In quick select, we only recursed on the side containing the target, but in quick sort, we want to recurse continuously on both sides until we every side contains only one element.

Worst case: $T(n) = n + T(n-1) = n + (n-1) + T(n-2) + \ldots = \Theta(n^2)$.

Best case: $T(n) = n + 2T\left(\frac{n}{2}\right) = \Theta(n \lg n)$.

Average case:

$$
T(n) = cn + \frac{1}{n}(T(n-1) + T(1)) + \frac{1}{n}(T(n-2) + T(2)) + \ldots + \frac{1}{n}(T(1) + T(n-1))
$$

$$
= \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1)) + \underbrace{\Theta(n)}_{\text{partition step}}
$$

$$
\leq \frac{1}{n}\sum_{i=0}^{n-1} 2\max\{T(i), T(n-i-1)\} + \Theta(n)
$$

$$
= \frac{1}{n} \cdot \frac{1}{2}(T(\frac{3i}{4})) + \frac{1}{n} \cdot \frac{1}{2}T(i) + cn
$$

Take note of two important observations here:

1. At each level of the recursion, the total work adds up to n (roughly).

2. This continues until we reach the end of the recursion at $n = 1$.

Every two levels, every block is half of its original size or less.

The worst case of quick sort is *extremely* rare. Alejandro ran an experiment, and out of one billion runs, the worst case took just twice the time of the average case.

Bubblesort is really bad. It's only taught because it has a cool name.

Mergesort cannot be improved upon, but quick sort can be improved by optimizing the algorithms in selecting pivots.

To review, quick sort operates with the following runtimes:

- Average case: $\Theta(n \lg n)$, with a constant ≈ 1.4, on a uniform input distribution. However, we know the input is *not* uniform. One type of typical distribution would be an array containing the old customer list (sorted) followed by a new customer list, and we'd want to merge the two lists. That is not a uniform distribution.

- Expected case: $\Theta(n \lg n)$ if we choose the pivot position at random. This is the best option in general.

- Worst case: $\Theta(n \lg n)$, with a constant $\approx\in [7, 10]$, if you select the median in $\Theta(n)$ time. This constant is relatively large, which is why we typically avoid this algorithm.

Note that the constants only matter when we can't improve the order any further, as is the case here.

Can we do better than $\Theta(n \lg n)$? It depends on what you're sorting. But in general, no, regardless of the algorithm (even considering other sorting algorithms other than quick sort). There is a lower bound for the performance of *all* sorting algorithms by $\Omega(n \lg n)$.

Finding the minimum (or maximum) element in a list or unsorted array of numbers has a lower bound of $\Omega(n)$ because you need to look at all n numbers. We can find a similar lower bound (and rationale) for sorting.

7.4   Sorting in the Comparison Model

Elements to be sorted can be compared but no other operations are allowed.

Theorem. In the comparison model, it takes $\Omega(n \lg n)$ operations to sort n objects.

Recall the 20 questions game from childhood. If you're unaware, 20 questions is a game where:

- A mineral, plant, or animal is selected.

- The other person asks 20 yes/no questions.

- The other person aims to guess the mineral/plant/animal that was selected, within those 20 questions.

For the guess to be correct, it can only be made in nodes where the set of candidates is of size 1. That is, you must narrow the number of possibilities down to 1 prior to making a guess in order for your guess to be correct.

If the tree of questions is balanced, we could distinguish 2^{20} objects since there are 2^{20} leaves (maximum) on a tree of height 20. If you have a completely unbalanced tree, you can only distinguish between 20 objects in a tree of height 20. This is one of the reasons why mergesort is good.

Fundamentally, shuffling and sorting are the same operation, just backwards. Every sorting sequence corresponds to a unique shuffle sequence. How many different shuffles do we have for a set that contains n elements? $n!$.

Let's consider a comparison-based sorting algorithm on a set where all the keys are distinct. We get a tree where the root is $a_i : a_j$ and the two child nodes are $a_i < a_j$ and $a_i > a_j$. As we get further down the tree, these comparisons all compound to narrow down the set. The algorithm determines the next question to ask.

$a_i : a_j$ is just one way of representing that we're doing a general comparison. Sometimes we also use the notation $a_i ? a_j$ or $a_i \gtrless a_j$.

At the end of the comparisons, we *must* have a single unique candidate shuffle, namely the sequence that is unshuffled by the algorithm as a result of those comparisons.

We need a tree with $n!$ leaves that is as shallow as possible. More specifically:

$$h = \log(n!) = \log\left(\frac{n^n}{e^n}\right) = \log n^n - \log e^n = n \log n - n \log e = \Theta\left(\left(\frac{n}{e}\right)^n\right)$$

This is true because of Stirling's approximation.

$\therefore$ As long as your sorting algorithm only uses comparison operators, $\Omega(n \lg n)$ is the best you can do.

When making an argument like this, you need to make arguments that apply across *all* algorithms, even ones that no one has seen or thought of yet. That's what we did here. When you're discussing lower bounds, you need to be extremely clear which operations are and aren't allowed.

7.5   Examples of Sorting Faster Than n lg n

- You are given a permutation of the numbers $1 \ldots n$. This runs in $\Theta(n)$ or $\Theta(1)$ time, depending on which operations you're allowed to use. If we're given the correct operations, we don't even need to look at the input in this case.

- Keys given belong to a set of size $O(n)$, allowing possible repetition.

Here's an algorithm for `counting_sort(A, k)`, where A is an array and k is the size of the universe (not the number of elements).

```
c[0 ...k - 1] = 0⃗; // Θ(n) = O(k)
for i = 0 to n - 1 do
    // Θ(n)
    c[A[i]]++;
end
for i = 1 to k - 1 do
    // Θ(n) < Θ(k)
    c[i] = c[i] + c[i - 1];
end
B = (copy A); // Θ(n)
for i = n - 1 down to 0 do
    // Θ(n)
    decrement C[B[i]];
    A[c[B[i]]] = B[i]
end
```

This works well when you have a dense set such as student IDs or street addresses.