CS240 Notes

Jacky Zhao

June 29, 2020

1 Course Objectives

1.1 Overview

What is this course about?

- When first learning to program, we emphasize correctness
- Starting with this course, we will also be converned with efficiency
- We will study efficient methods of storing, accessing, and performing operations on large collections of data.
- Typical operations include: inserting new data items, deleting data items, searching for specific data items, sorting
- We will consider various abstract data types (ADTs) and how to implemnet them efficiently using appropriate data structures.
- There is a strong emphasis on mathematical analysis in the course
- Algorithms are presented using pseudocode and analyzed using order notation (big-O, etc.)

Course Topics:

- big-O analysis
- priority queues and heaps
- sorting, selection
- binary search trees, AVL trees, B-trees
- skip lists
- hashing
- quadtrees, kd-trees
- range search
- tries
- string matching
- data compression

Required knowledge:

- arrays, linked lists (3.2- 3.4)
- strings (3.6)
- stacks, queues (4.2 - 4.6)
- abstract data types (4 - intro, 4.1, 4.8 - 4.9)
- recursie algorithms (5.1)
- binary trees (5.4 - 5.7)
- sorting (6.1 - 6.4)
- binary search (12.4)
- binary search trees (12.5)
- probability and expectations

1.2 General Terminologies

The core of CS240 is:

Given problem Π, design algorithm A that solves it, and analyze its efficiency

So what is a problem, an algorithms, and how do you quantify efficiency?

Problem

- Given a problem instance, carry out a particular computational task
- Ex. Sorting is a problem

Problem Instance

- Input for the specified problem

Problem Solution

- Output (correct answer) for the specified problem instance

Size of a problem instance

- $Size(I)$ is a positive integer which is a measure of the size of the instance I

Algorithm

- a step-by-step process (e.g. described in pseudocode) for carrying out a series of computations, given an arbitrary problem instance I

Algorithm solving a problem

- an algorithm A solves a problem Π if, for every instance I of Π, A finds (computes) a valid solution for the instance I in finite time

Program

- an implementation of an algorithm using a specified computer language

Pseudocode

- a method of communicating an algorithm to another person
- in contrast, a program is a method of communicating an algorithm to a computer
- General rules of pseudocode:
  - omits obvious details (variable declarations)
  - has limited, if any, error detection
  - sometimes uses English descriptions
  - sometimes usus mathematical notation

1.3 Algorithms and programs

For a problem Π, we can have several algorithms.
For an algorithm A solving Π, we can have several programs (implementations)

Algorithms in practice: Given a problem Π:

1. **Algorithm Design:** Design an algorithm A that solves Π
2. **Algorithm Analysis:** Assess correctness and efficiency of A
3. If acceptable (correct and efficient), implement A.

2 Analysis of Algorithms I

- **Running Time:** In this course, we are primarily concerned with the amount of time a program takes to run
- **Space:** We also may be interested in the amount of memory the program requires
- The amount of time and/or memory required by a program will depend on $Size(I)$, the size of the given problem instance I

2.1 Running time of Algorithms/Programs

Option 1: Experimental Studies

- Write a program implementing the algorithm
- Run the programs with various sizes of input and measure the actual running time
- Plot/compare the results

Shortcomings:

- Implementation may be complicated/costly
- Timings are affected by many factors: hardware, software environment, and human factors
- We cannot test all inputs (what are good sample inputs?)
- We cannot easily compare two algorithms/programs

We want a framework that:

- Does not require implementing the algorithm
- Is independent of the hardware/software environment
- Takes into account all input instances

Which means, we need some simplifications

We will develop several aspects of algorithm analysis:

- Algorithms are presented in structured high-level pseudocode, which is language-independent
- Analysis of algorithms is based on an idealized computer model
- The efficiency of an algorithm (with respect to time) is measure din terms of its growth rate, aka the complexity of the algorithm

2.2 Simplifications of running time

Overcome dependency on hardware/software

- Express algorithms using pseudocode
- Instead of time, count the number of primitive operations
- Implicit assumption: primitive operations have fairly similar, though different, running time on different systems

Random Access Machine (RAM) model:

- it has a set of memory cells, each of which stores one item (word) of data
- any access to a memory location takes constant time
- any primitive operation takes constant time
- the running time of a program can be computed to be the number of memory accesses plus the number of primitive operations

This is an idealized model, so these assumptions may not be valid for a "real" computer

Simplify Comparisons

- Example: Compare $100n$ with $10n^2$
- Idea: Use order notation
- Informally: ignore constants and lower order terms

We will simplify our analysis by considering the behaviour of algorithms for large input sizes

2.3 Asymptotic Notation

O-notation

- $f(n) \in O(g(n))$ if there exist constants $C > 0$ and $n_0 > 0$ such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$
- Example: $f(n) = 75n + 500$ and $g(n) = 5n^2$, choose $c = 1$ and $n_0 = 20$ can prove $f(n) \in O(g(n))$
- Note: the absolute value signs int eh definition are irrelevant for analysis of run-time or space, but are useful in other application sof asymptotic notation

Example of Order Notation:
In order to prove that $2n^2 + 3n + 11 \in O(n^2)$ from first principles, we need to find c and n_0 such that:

$$0 \leq 2n^2 + 3n + 11 \leq cn^2 \text{ for all } n \geq n_0$$

Note that all choices of c and n_0 will work. **Solution:**

Choose $n_0 = 1$.

$$n_0 \leq n \to 1 \leq n \to 1 \leq n^2 \to 11 \leq 11n^2$$
$$n_0 \leq n \to 1 \leq n \to n \leq n^2 \to 3n \leq 3n^2$$
$$\text{We also have: } 2n^2 \leq 2n^2$$

So we have:

$$2n^2 + 3n + 11 \leq 2n^2 + 3n^2 + 11n^2 \leq 16n^2$$

So let $c = 16$ and $n_0 = 1$, and we have $|f(n)| < c|g(n)|$ for all $n \geq n_0$.
Thus $2n^2 + 3n + 11 \in O(n^2)$. □

We want a **tight** asymptotic bound. So we have:
Ω-notation

- $f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $c|g(n)| \leq |f(n)|$ for all $n \geq n_0$

Θ-notation

- $f(n) \in \Theta(g(n))$ if there exist constants $c_1, c_2 > 0$, and $n_0 > 0$ such that $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ for all $n \geq n_0$

Notice:

$$f(n) \in \Theta(g(n)) \longleftrightarrow f(n) \in O(g(n)) \textbf{ and } f(n) \in \Omega(g(n))$$

Example:
Prove that $\frac{1}{2}n^2 - 5n \in \Omega(n^2)$ from first principles.

Solution:
Let $n_0 = 20$. We find c.

$$
\begin{aligned}
n_0 = 20 \leq n \rightarrow 20n \leq n^2 \rightarrow 5n \leq \frac{1}{4}n^2 \rightarrow 0 \leq \frac{1}{4}n^2 - 5n \\
\frac{1}{2}n^2 - 5n = \frac{1}{4}n^2 + \underbrace{\frac{1}{4}n^2 - 5n}_{\geq 0} \geq \frac{1}{4}n^2
\end{aligned}
$$

Since $\frac{1}{2}n^2 - 5n \geq \frac{1}{4}n^2$, we choose $c = \frac{1}{4}$ and we have $\frac{1}{2}n^2 - 5n \in \Omega(n^2)$. $\square$

Quick Summary:

- $O \leftrightarrow$ asymptotically not bigger
- $\Omega \leftrightarrow$ asymptotically not smaller
- $\Theta \leftrightarrow$ asymptotically the same

We have $f(n) = 2n^2 + 3n + 11 \in \Theta(n^2)$

- How do we express that $f(n)$ is **asymptotically strictly smaller** than n^3?

o-notation

- $f(n) \in o(g(n))$ if for **all** constants $c > 0$, there exists a constant $n_0 > 0$ such that $|f(n)| < c|g(n)|$ for all $n \geq n_0$

ω-notation

- $f(n) \in \omega(g(n))$ if for **all** constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq c|g(n)| < |f(n)|$ for all $n \geq n_0$

The o and ω notations are rarely proved from first principles.

2.4 Relationships between Order Notations

- $f(n) \in \Theta(g(n)) \leftrightarrow g(n) \in \Theta(f(n))$
- $f(n) \in O(g(n)) \leftrightarrow g(n) \in \Omega(f(n))$
- $f(n) \in o(g(n)) \leftrightarrow g(n) \in \omega(f(n))$

- $f(n) \in o(g(n)) \rightarrow f(n) \in O(g(n))$
- $f(n) \in o(g(n)) \rightarrow f(n) \notin \Omega(g(n))$
- $f(n) \in \omega(g(n)) \rightarrow f(n) \in \Omega(g(n))$
- $f(n) \in \omega(g(n)) \rightarrow f(n) \notin O(g(n))$

2.5 Algebra of Order Notations

Identity rule

- $f(n) \in \Theta(f(n))$

Maximum rules

Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$, then:

- $O(f(n) + g(n)) = O(max\{f(n), g(n)\})$
- $\Omega(f(n) + g(n)) = \Omega(max\{f(n), g(n)\})$

Transitivity

- if $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$
- if $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n))$, then $f(n) \in \Omega(h(n))$

2.6 Techniques for Order Notation

Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n > n_0$. Suppose that:

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$$

Then

$$f(n) \in \begin{cases} o(g(n)) & \text{if } L = 0 \\ \Theta(g(n)) & \text{if } 0 < L < \infty \\ \omega(g(n)) & \text{if } L = \infty \end{cases}$$

The required can often be computed using *l'Hôpital's rule*.
Note that this result gives sufficient (but not necessary) conditions for the stated conclusions to hold.

Example1:
Let $f(n)$ be a polynomial of degree $d \geq 0$

$$f(n) = c_d n^d + c_{d-1} n^{d-1} + \cdots + c_1 n + c_0$$

for some $c_d > 0$.
Then $f(n) \in \Theta(n^d)$.

Solution:

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{n^d} &= \lim_{n\to\infty} \frac{(c_d n^d + c_{d-1} n^{d-1} + \cdots + c_1 n + c_0)'}{(n^d)'} \\
&= \lim_{n\to\infty} \frac{(c_d)(d)n^{d-1} + (c_{d-1})(d-1)n^{d-2} + \cdots + (c_1)(1)n^0 + 0)'}{dn^{d-1}} \\
&= \lim_{n\to\infty} \frac{(c_d)(d)n^{d-1}}{dn^{d-1}} + \lim_{n\to\infty} \frac{(c_{d-1})(d-1)n^{d-2}}{dn^{d-1}} + \cdots + \lim_{n\to\infty} \frac{(c_1)(1)n^0}{dn^{d-1}} \\
&= \lim_{n\to\infty} \frac{(c_d)(d)n^{d-1}}{dn^{d-1}} \\
&= \lim_{n\to\infty} c_d \\
&= c_d
\end{aligned}$$

Since $c_d > 0$, we know that $f(n) \in \Theta(n^d)$, as desired. $\square$

Example2:
Prove that $f(n) = n(2 + \sin(\frac{n\pi}{2}))$ is $\Theta(n)$.
Note that $\lim_{n\to\infty}(2 + \sin(\frac{n\pi}{2}))$ does not exist.

Solution:

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{n(2 + \sin(\frac{n\pi}{2}))}{n} \\
&= \underbrace{\lim_{n\to\infty} (2 + \sin(\frac{n\pi}{2}))}_{\text{DNE, no conclusion}}
\end{aligned}$$

Think another way:

$$\begin{aligned}
-1 \leq \sin(\frac{n\pi}{2}) \leq 1 \\
1 \leq 2 + \sin(\frac{n\pi}{2}) \leq 3 \\
\text{Let } n_0 = 1, \text{ so } n \geq 1 \\
1n \leq 2 + \sin(\frac{n\pi}{2}) \leq 3n
\end{aligned}$$

So we have $n_0 = 1$, $c_1 = 1$ and $c_0 = 1$. And thus $n(2 + \sin(\frac{n\pi}{2})) \in \Theta(n)$ $\square$

2.7 Growth Rates

- If $f(n) \in \Theta(g(n))$, then the growth rates of $f(n)$ and $g(n)$ are the same
- If $f(n) \in o(g(n))$, then the growth rates of $f(n)$ is less than $g(n)$
- If $f(n) \in \omega(g(n))$, then the growth rates of $f(n)$ is greater than $g(n)$
- Typically, $f(n)$ may be complicated and $g(n)$ is chosen to be a very simple function

Example3:
Compare the growth rates of $\log n$ and n.
Note: In this course, we default the base of log to be 2, so by $\log n$ we mean $\log_2 n$

Solution:

$$\begin{aligned}\lim_{n\to\infty} \frac{\log n}{n} &\overset{H}{=} \lim_{n\to\infty} \frac{\frac{1}{n\ln 2}}{1} \\ &= \lim_{n\to\infty} \frac{1}{n\ln 2} \\ &= 0\end{aligned}$$

So $\log n \in o(n)$ and

Now compare the growth rates of $(\log n)^c$ and n^d, where $c, d > 0$ are arbitrary numbers.

$$\begin{aligned}\lim_{n\to\infty} \frac{(\log n)^c}{n^d} &\overset{H}{=} \lim_{n\to\infty} \frac{c(\log n)^{c-1}\frac{1}{n\ln 2}}{dn^{d-1}} \\ &= \lim_{n\to\infty} \frac{c(\log n)^{c-1}}{d(\ln 2)n^d} \\ &\overset{H}{=} \lim_{n\to\infty} \frac{c(c-1)(\log n)^{c-2}}{d^2(\ln 2)^2 n^d} \\ &\overset{H}{=} \dots \\ &\overset{H}{=} \lim_{n\to\infty} \frac{c!}{(\ln 2)^c d^c n^d} \\ &= 0\end{aligned}$$

So $(\log n)^c \in o(n^d)$, meaning $(\log n)^c$ has growth rate less than n^d for arbitrary $c, d > 0$.
This means, even if we have $(\log n)^1 0000$, we will still have a growth rate less than n^2

2.8 Common Growth Rates

Commonly encountered growth rates in analysis of algorithms include the following (in increasing order of growth rate):

- $\Theta(1)$ (constant complexity)
- $\Theta(\log n)$ (logarithmic complexity)
- $\Theta(n)$ (linear complexity)
- $\Theta(n \log n)$ (linearithmic)
- $\Theta(n \log^k n)$ for some constant k (quasi-linear)
- $\Theta(n^2)$ (quadratic complexity)
- $\Theta(n^3)$ (cubic complexity)
- $\Theta(2^n)$ (exponencial complexity)

It is interesting to see how the running time is affected when the size of the problem instance doubles (i.e. $n \to 2n$)

constant complexity:	$T(n) = c$	$\rightarrow$	$T(2n) = c$
logarithmic complexity:	$T(n) = c \log n$	$\rightarrow$	$T(2n) = T(n) + c$
linear complexity:	$T(n) = cn$	$\rightarrow$	$T(2n) = 2T(n)$
linearithmic:	$T(n) = cn \log n$	$\rightarrow$	$T(2n) = 2T(n) + 2cn$
quadratic complexity:	$T(n) = cn^2$	$\rightarrow$	$T(2n) = 4T(n)$
cubic complexity:	$T(n) = cn^3$	$\rightarrow$	$T(2n) = 8T(n)$
exponencial complexity:	$T(n) = c2^n$	$\rightarrow$	$T(2n) = \frac{T(n)^2}{c}$

2.9 Techniques for Algorithm Analysis

Goal: Use asymptotic notation to simplify run-time analysis

- running time of an algorithm depends on the input size n
- identify elementary operations that require $\Theta(1)$ time
- the complexity of a loop is expressed as the sum of the complexities of each iteration of the loop
- Nested loops: starts with the innermost loop and proceed outwards.
  This gives nested summations

Example:

Test1

```
sum ← 0
for i ← 1 to n do
    for j ← i to n do
        sum ← sum + (i + j)^2
return sum
```

We have:

$$\begin{aligned}
T(n) &= c_0 + c_1 + \sum_{i=1}^{n} \sum_{j=i}^{n} c_2 \\
&= c_0 + c_1 + \sum_{i=1}^{n} c_2(n - i + 1) \\
&= c_0 + c_1 + \sum_{i=1}^{n} c_2 n - \sum_{i=1}^{n} c_2 i + \sum_{i=1}^{n} c_2 \\
&= c_0 + c_1 + c_2 n^2 - c_2(\frac{n(n+1)}{2}) + c_2 n \\
&= c_0 + c_1 + c_2(n^2 - \frac{n^2 - n}{2} + n) \\
&= c_0 + c_1 + \frac{c_2}{2}(n^2 + n)
\end{aligned}$$

So $T(n) \in \Theta(n^2)$

Another way of doing the same thing is to find upper bound and lower bound.

$$
\begin{aligned}
T(n) &= c_0 + c_1 + \sum_{i=1}^{n}\sum_{j=i}^{n} c_2 \leq c_0 + c_1 + \sum_{i=1}^{n}\sum_{j=1}^{n} c_2 \\
&= c_0 + c_1 + c_2 \sum_{i=1}^{n}\sum_{j=1}^{n} 1 \\
&= c_0 + c_1 + c_2 n^2
\end{aligned}
$$

So $T(n) \in O(n^2)$

$$
\begin{aligned}
T(n) &= c_0 + c_1 + \sum_{i=1}^{n}\sum_{j=i}^{n} c_2 \geq c_0 + c_1 + \sum_{i=1}^{n/2}\sum_{j=1}^{n} c_2 \\
&\geq c_0 + c_1 + \sum_{i=1}^{n/2}\sum_{j=n/2+1}^{n} c_2 \\
&= c_0 + c_1 + \sum_{i=1}^{n/2} c_2 \frac{n}{2} \\
&= c_0 + c_1 + c_2 \frac{n}{2} \sum_{i=1}^{n/2} 1 \\
&= c_0 + c_1 + c_2 (\frac{n}{2})(\frac{n}{2}) \\
&= c_0 + c_1 + c_2 (\frac{n^2}{4})
\end{aligned}
$$

So $T(n) \in \Omega(n^2)$. Therefore $T(n) \in \Theta(n^2)$

Two general strategies are as follows:

- Use Θ-bounds throughout the analysis and obtain a Θ-bound for the complexity of the algorithm
- Prove a O-bound and a matching Ω-bound separately.
  Use upper bounds (for O-bounds) and lower bounds (for Ω-bounds) early and frequently
  This may be easier because upper/lower bounds are easier to sum.

2.10 Complexity of Algorithms

Algorithm can have different running times on two instances of the same size

Test3(A, n)
A: array of size n
for $i \leftarrow 1$ to $n-1$ do
  $j \leftarrow i$
  while $j > 0$ and $A[j] > A[j-1]$ do
    swap $A[j]$ and $A[j-1]$
    $j \leftarrow j-1$

Let $T_A(I)$ denote the running time of an algorithm A on instance I.

Worst-case complexity of an algorithm

- it is a function $f : \mathbb{Z}^+ \to \mathbb{R}$ mapping n (the input size) to the longest running time for any input instance of size n

$$T_A(n) = max\{T_A(I) : Size(I) = n\}$$

Average-case complexity of an algorithm

- it is a function $f : \mathbb{Z}^+ \to \mathbb{R}$ mapping n (the input size) to the average running time of A over all instances of size n

$$T_A^{avg}(n) = \frac{1}{|\{I : Size(I) = n\}|} \sum_{I:Size(I)=n} T_A(I)$$

The average is more important in real life, but it is also harder to calculate.
In this course, we are talking about **worst case** complexity by default.

We need to convince/explain why a case is the worst case and compute its running time.

In the example of Test3 above, the worst number of times the while loop will run is i times.
So $\sum_{i=1}^{n-1} ic \in \Theta(n^2)$.
Note that the average running time for this code is also $\Theta(n^2)$.

2.11 O-notation and Complexity of Algorithms

We should not compare complexity of algorithms using O-notation because:

- the worst-case run-time may only be achieved on some instances
- O-notation is an upper bound

So if we want to compare algorithms, we should always use Θ-notation.

2.12 Analysis of Merge Sort

Design of Merge Sort
Input: Array A of n integers

- Step 1: We split A into two sub-arrays: A_L consists of the first $\lceil \frac{n}{2} \rceil$ elements in A and A_R consists of the last $\lfloor \frac{n}{2} \rfloor$ elements in A
- Step 2: Recursively run MergeSort on A_L and A_R
- Step 3: After A_L and A_R have been sorted, use a function Merge to merge them into a single sorted array

MergeSort implementation

MergeSort($A, l \leftarrow 0, r \leftarrow n-1$)
A: array of size n, $0 \leq l \leq r \leq n-1$

```
if (r ≤ l) then
   return
else
   m = (r + l)/2
   MergeSort(A, l, m)
   MergeSort(A, m+1, r)
   Merge(A, l, m, r)
```

$$T(n) = 2T(\frac{n}{2}) + \Theta(Merge)$$

Merge implementation

Merge(A, l, m, r)
$A[0 \dots n-1]$ is an array, $A[l \dots m]$ is sorted, $A[m+1 \dots r]$ is sorted

```
initialize auxiliary array S[0...n-1]
copy A[l...r] into S[l...r]
int i_L <- l; itn i_R <- m+1;
for (k <- l; k <= r; k++) do
    if(i_L > m) A[k] <- S[i_R++]
    else if (i_R > r) A[k] <- S[i_L++]
    else if (S[i_L] <= S[i_R]) A[k] <- S[i_L++]
    else A[k] <- S[i_R++]
```

So Merge takes time $\Theta(r-l+1)$, which is $\Theta(n)$ time for merging n elements
Therefore the overall running time of MergeSort is:

$$T(n) = 2T(\frac{n}{2}) + \Theta(n)$$

Analysis of MergeSort:
Let $T(n)$ denote the time to run MergeSort on an array of length n

- Step 1 takes time $\Theta(n)$
- Step 2 takes time $T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$
- Step 3 takes time $\Theta(n)$

The recurrence relation for $T(n)$ is as follows:

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

It suffices to consider the following exact recurrence, with constant factor c replacing Θ's:

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

The following is the corresponding sloppy recurrence
(meaning it has floors and ceilings removed)

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + cn & \text{if } n > 1 \\ c & \text{if } n = 1 \end{cases}$$

The exact and sloop recurrences are identical when n is a power of 2
The recurrence can easily be solved by various methods when $n = 2^j$
The solution has growth rate $T(n) \in \Theta(n \log n)$
It is impossible to show that $T(n) \in \Theta(n \log n)$ for all n by analyzing the exact recurrence.

So how to show $T(n) \in \Theta(n \log n)$ when $n = 2^j$?

$$\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + cn \\
&= 2(2T(\frac{n}{2^2}) + c\frac{n}{2}) + cn \\
&= 2^2T(\frac{n}{2^2}) + 2cn \\
&= 2^2(T(\frac{n}{2^3}) + c\frac{n}{2^2}) + 2cn \\
&= 2^3T(\frac{n}{2^3}) + 3cn \\
&\dots \\
&= 2^jT(\frac{n}{2^j}) + jcn \\
&= 2^jc + jcn \\
&= cn + jcn \\
&= cn + cn\log n
\end{aligned}$$

So $T(n) \in \Theta(n \log n)$

Here's another way of proving it:
We know that $T(n) = 2T(\frac{n}{2}) + cn$.
So $T(\frac{n}{2}) = 2T(\frac{n}{4}) + c\frac{n}{2}$
We can draw a tree of the cost of $T(n)$, $T(\frac{n}{2})$, $T(\frac{n}{4})$ and more, and sum them.
So we get $T(n) = (\log n + 1)cn = cn\log n + cn \in \Theta(n \log n)$

2.13 Common Recurrence Relations

Recursion	Resolves to	Example
$T(n) = T(\frac{n}{2}) + \Theta(1)$	$T(n) \in \Theta(\log n)$	Binary search
$T(n) = 2T(\frac{n}{2}) + \Theta(n)$	$T(n) \in \Theta(n \log n)$	Merge sort
$T(n) = 2T(\frac{n}{2}) + \Theta(\log n)$	$T(n) \in \Theta(n)$	Heapify
$T(n) = T(cn) + \Theta(n)$ for some $0 < c < 1$	$T(n) \in \Theta(n)$	Selection
$T(n) = 2T(\frac{n}{4}) + \Theta(1)$	$T(n) \in \Theta(\sqrt{n})$	Range Search
$T(n) = T(\sqrt{n}) + \Theta(1)$	$T(n) \in \Theta(\log\log n)$	Interpolation Search

Once you know the result, it is usually easy to prove by induction
Many more recursions, and some methods to find the result, in CS341

2.14 Summary & Helpful formulas

O-notation

- $f(n) \in O(g(n))$ if there exist constants $C > 0$ and $n_0 > 0$ such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$

Ω-notation

- $f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $c|g(n)| \leq |f(n)|$ for all $n \geq n_0$

Θ-notation

- $f(n) \in \Theta(g(n))$ if there exist constants $c_1, c_2 > 0$, and $n_0 > 0$ such that $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ for all $n \geq n_0$

o-notation

- $f(n) \in o(g(n))$ if for **all** constants $c > 0$, there exists a constant $n_0 > 0$ such that $|f(n)| < c|g(n)|$ for all $n \geq n_0$

ω-notation

- $f(n) \in \omega(g(n))$ if for **all** constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq c|g(n)| < |f(n)|$ for all $n \geq n_0$

Useful Sums:

Arithmetic sequence

$$\sum_{i=0}^{n-1}(a+di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2) \text{ if } d \neq 0$$

Geometric sequence

$$\sum_{i=0}^{n-1} ar^i = \begin{cases} a\frac{r^n-1}{r-1} & \in \Theta(r^n) & \text{if } r > 1 \\ na & \in \Theta(n) & \text{if } r = 1 \\ a\frac{1-r^n}{1-r} & \in \Theta(1) & \text{if } 0 < r < 1 \end{cases}$$

Harmonic sequence

$$\sum_{i=0}^{n-1} \frac{1}{i} = \ln n + \gamma + o(1) \in \Theta(\log n)$$

A few more

$$\sum_{i=0}^{n-1} \frac{1}{i^2} = \frac{\pi^2}{6} \in \Theta(1)$$

$$\sum_{i=0}^{n-1} i^k \in \Theta(n^k + 1) \text{ for } k \geq 0$$

2.15 Useful Math Facts

Logarithms

- $a^{\log_b c} = c^{\log_b a}$
- $\frac{d}{dx} \ln x = \frac{1}{x}$

Factorial

- $n!$ = number of ways to permute n elements
- $\log(n!) = \log n + \log(n-1) + \cdots + \log 1 \in \Theta(n \log n)$

Probability and moments

- (linearity of expectation)
- $E[aX] = aE[X]$
- $E[X+Y] = E[X] + E[Y]$

3 Heap

3.1 Abstract Data Types

ADT

- A description of information and a collection of operations on that information
- The information is accessed only through the operations

We have various realizations of an ADT, which specify:

- how the information is stored (Data Structure)
- how the operations are performed (Algorithms)

3.2 Stack ADT

Stack

- An ADT consisting of a collection of items with operations:
  - push: inserting an item
  - pop: removing the most recently inserted item
- Items are removed in LIFO order
- Items enter the stack at the top and are removed from the top
- We can have extra operations: size, isEmpty, and top

Applications

- addresses of recently visited websites
- procedure calls

Realizations of Stack ADT

- using arrays
- using linked lists

3.3 Queue ADT

Queue

- and ADT consisting of a collection of items with operations
  - enqueue: inserting an
  - dequeue: removing the last recently inserted item
- items are removed in FIFO order
- items enter the queue at the rear and are removed from the front
- we can have extra operations: size, isEmpty, and front

Applications

- waiting lines
- printer queues

Realizations of Queue ADT

- using (circular) arrays
- using linked lists

3.4 Priority Queue ADT

Priority Queue

- An ADT consisting of a collection of items (each having a priority) with operations
  - insert: inserting and item tagged with a priority
  - deleteMax: removing the item of highest priority
- deleteMax is also called extractMax or getMax
- the priority is also called key

The above definition is for a maximum-oriented priority queue.
A minimum-oriented priority queue is defined in the natural way, replacing operation deleteMax by deleteMin

Applications

- typical todo list
- simulation systems
- sorting

Using a PQ to sort

PQ-Sort $(A[0 \ldots n-1])$
    initialize PQ to an empty priority queue
    for $k \leftarrow 0$ to $n-1$ do
        PQ.insert(A[k], A[k])   (priority and item are equal to A[k])
    for $k \leftarrow n-1$ down to 0 do
        $A[k] \leftarrow PQ.deleteMax()$

- runtime $O(\sum_{0\leq i<n} insert(i) + \sum_{0\leq i<n} deleteMax(i))$
- depends on how we implement hte priority queue

3.5 Realizations of Priority Queues

Realization 1: unsorted arrays

- insert: $O(1)$ (append to end)
- deleteMax: $O(n)$ (linear search for max)

Note: we assume dynamic arryas. i.e. expand by doubling as needed.
Amortized over all insertions this takes $O(1)$ extra time.
Proof:
Suppose we start from A of length 1. We do n insert, $n = 2^k$

$$\begin{aligned}
\text{Total cost of inserts } &= O(\underbrace{1+1+\cdots+1}_{\text{n times}} + \underbrace{1+2+4+8+\cdots+2^{k-1}}_{2^k-1=n-1}) \\
&= O(2n-1) \\
&= O(n)
\end{aligned}$$

Using unsorted linked lists is identical.
PQ-sort with this realization yields selection sort, so runtime is:

$$O(\sum_{i<n} i) = O(n^2)$$

Realization 2: sorted arrays

- insert: $O(n)$
- deleteMax: $O(1)$

Using sorted linked lists is identical.
PQ-sort with this realization yields insertion sort, runtime is:

$$O(\sum_{i<n} i) = O(n^2)$$

3.6 Heaps (binary)

Binary heap

- is a certain type of binary tree
- Recall a few things:
  - a binary tree is either
    - empty, or
    - consist of three parts:
      a node & 2 binary trees (left subtree and right subtree)
  - few terms: root, leaf, parent, child, level sibling, ancestor, descendant, etc.
  - any binary tree with n nodes has height at least
    $\log(n+1) - 1 \in \Omega(\log n)$
- Also remember that the height of a non-empty tree is the length of the longest path from root to node
- The height of the empty tree is -1

Heap

- is a binary tree with the following two properties
  - Structural Property: All the levels of a heap are completely filled, except)possibly) for the last level. The filled items in the last level are left-justified.
  - Heap-order Property: For any node i, the key of the parent of i is larger than or equal to key of i

The full name for this is max-oriented binary heap **Lemma:** The height of a heap with n nodes is $\Theta(\log n)$

Storing Heaps in Arrays

- **Heaps should not be stored as binary trees!**
- Let H be a heap of n items and let A be an array of size n. Store root in $A[0]$ and continue with elements level-by-level from top to bottom, in each level left-to-right

It is easy to navigate the heap using this array representation:

- the root node is at index 0
- the left child of node i (if it exists) is node $2i + 1$
- the right child of node i (if it exists) is node $2i + 2$
- the parent of node i (if it exists) is node $\lfloor \frac{i-1}{2} \rfloor$
- the last node is $n - 1$

PAUSED, Speed run for midterm