CS-457 : Notes

Charlie Stuart : src322

Fall 2021

Note: I made up the seciton order because I'm quirky

Contents

1 Other Resources **2**

2 Math Review **3**

2.1 Set Theory . . . 3

2.2 Functions . . . 4

2.3 Summations . . . 4

2.4 Limits . . . 6

2.5 Logarithm Rules . . . 6

2.6 Probability . . . 6

3 Algorithms Introduction **7**

4 Run-Time **8**

4.1 Asymptotic Notation . . . 8

4.1.1 Table of Informal Definitions . . . 8

4.1.2 Table of Formal Definitions . . . 9

4.1.3 Table of Limit Definitions . . . 9

4.1.4 Properties . . . 9

4.2 Recurrence Relations . . . 9

4.2.1 Divide and Conquer Algorithms . . . 9

4.3 General Form . . . 10

4.4 Recursion Tree . . . 10

4.4.1 Solving . . . 10

4.5 Substitution . . . 11

4.6 Master Theorem . . . 11

4.6.1 Replacement of Variables . . . 12

5 Expected Runtime - Randomization **14**

5.1 Quick Sort Example . . . 14

5.2 Probabilistic Analysis . . . 15

5.3 Linearity of Expectation . . . 16

6 Hashing **17**

7 Binary Trees **20**

7.1 Search Trees . . . 20

7.2 Heaps . . . 22

7.3 **RedBlack** Trees . . . 24

7.4 Structural Induction . . . 25

8 In Class Examples **26**

8.1 L01 : Maximum Subarray Problem . . . 26

9 Dynamic Programming **28**

10 Graphs **29**

1 Other Resources

Algorithm Visualization : `https://www.cs.usfca.edu/%7Egalles/visualization/Algorithms.html`

Red/Black Tree Visualization : `https://www.cs.usfca.edu/~galles/visualization/RedBlack.html`

2 Math Review

2.1 Set Theory

From pages 3-7 in *Introduction to the Theory of Computation* by Michael Sipser

Set : A group of objects represented as a unit
Element : An object in a set
Member : An object in a set
Multi Set : An set containing an element that occurs multiple times
Subset : A set that consists of elements that exist in a different set
Proper Subset : A set that is a subset of another set, but not equal
Infinite Set : A set of infinitely many elements
Empty Set : A set of no elements
Singleton Set : A set of one elements
Unordered Pair : A set of two elements
Sequence : A set in a specific order
Tuple : A finite set
k-Tuple : A tuple of k elements
Ordered Pair : A 2-tuple
Power Set : All the subsets of A
$\in$: Is a member of
$\notin$: Is not a member of
$\subset$: Is a proper subset of
$\not\subset$: Is not a proper subset of
$\subseteq$: Is a subset of
$\not\subseteq$: Is not a subset of
$\cup$: Union of two sets
$\cap$: Intersection of two sets
$\times$: Cross product of two sets
$\mathbb{N}$: Set of natural numbers
$\mathbb{Z}$: Set of integers
$\mathbb{Q}$: Set of rational numbers
$\mathbb{A}$: Set of algebraic numbers
$\mathbb{R}$: Set of real numbers

A set is defined in a few ways

$S = \{7, 21, 57\}$	Finite Set
$S = \{1, 2, 3...\}$	Infinite Set of all natural numbers $\mathbb{N}$
$S = \{7, 7, 21, 57\}$	Multi Set
$S = \emptyset$	Empty Set
$S = \{5\}$	Singleton Set
$S = \{5, 3\}$	Unordered pair
$S = \{n \mid n = m^2 \text{ for some } m \in \mathbb{N}\}$	Set of perfect squares

The union of two sets is the same as an OR operator in boolean algebra. It's all the elements in both sets.

$$A = \{1, 2, 3\}$$
$$B = \{3, 4, 5\}$$
$$A \cup B = \{1, 2, 3, 4, 5\}$$

The intersection of two sets is the same as an AND operator in boolean algebra. It's all the elements that appear only in both sets.

$$A = \{1, 2, 3\}$$
$$B = \{3, 4, 5\}$$
$$A \cap B = \{3\}$$

The Cartesian product, or cross product, of two sets is the set of all ordered pairs where the first element is a member of the first set and the second element is a member of the second set for every combination.

$$A = \{1, 2\}$$
$$B = \{x, y, z\}$$
$$A \times B = \{(1, x), (2, x), (1, y), (2, y), (1, z), (2, z)\}$$

2.2 Functions

From pages 7-8 in *Introduction to the Theory of Computation* by Michael Sipser

Function : An objects that sets up an input-output relationship
Domain : The set of possible inputs to a function
Range : The set of possible outputs to a function

$f : D \to R$ Function f has domain D and range R

2.3 Summations

From CLRS Appendix A

REMEMBER : Summations are inclusive

Constants can be "taken out":

$$\sum_{i=1}^{n} cx_i = c\sum_{i=1}^{n} x_i$$

Addition can be broken up:

$$\sum_{i=1}^{n}(x_i + y_i) = \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} y_i$$

Arithmetic Series :

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

Sum of Squares :

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Sum of Cubes :

$$\sum_{i=0}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

Geometric Series : When $x \neq 1$ and is real

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + ... + x^n$$
$$\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}$$

Geometric Series : When the summation is infinite and $|x| < 1$

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1 - x}$$

Harmonic Series :

$$
\begin{aligned}
H(n) &= 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n} \\
H(n) &= \sum_{i=1}^{n} \frac{1}{i} \\
H(n) &= \ln n + O(1)
\end{aligned}
$$

Logarithms :

$$
\begin{aligned}
S(n) &= \sum_{i=1}^{n} \log(i) \\
S(n) &= \log(1) + \log(2) + ... + \log(n-1) + \log(n) \\
S(n) &= \log(1 * 2 * ... * (n-1) * n) \\
S(n) &= \log(n!)
\end{aligned}
$$

2.4 Limits

Indeterminate Forms : $\frac{\pm\infty}{\pm\infty}$, $\frac{0}{0}$

2.5 Logarithm Rules

$$
\begin{aligned}
\log_b(XY) &= \log_b(X) + \log_b(Y) \\
\log_b(\frac{X}{Y}) &= \log_b(X) - \log_b(Y) \\
\log_b(X^y) &= y \log_b(X)
\end{aligned}
$$

2.6 Probability

3 Algorithms Introduction

What is an algorithm?

1. A well-defined *computational procedure* that takes in *input* and produces *output*
2. A well-defined *sequence of computational steps* that transform the *input* into the *output*

Goal : Solve a *computational problem*

Correct Algorithms vs *Efficient* Algorithms:

- **Correct :** Computes the desired output on every problem instance
- **Efficient :** Time efficient, Quick to run
- **Efficient :** Space efficient, Memory usage
- **Efficient :** Amenable to parallelism, can be broken into subproblems
- **Efficient :** Not consuming too much bandwidth
- **Efficient :** Easy to write, code, remember

4 Run-Time

We measure the time efficiency of an algorithm with run-time

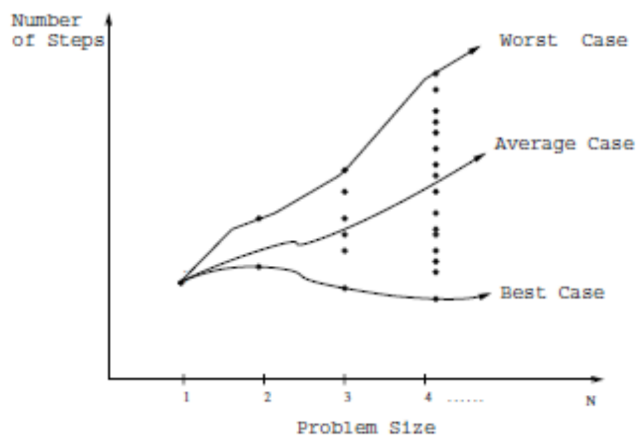
Actual run-time depends on:

1. The specs of the machine being used
2. The problem instance
3. The data structures used by the algorithm

We compare algorithms by:

- Code and run experiments
- Analyze run-time as a function of input

Worst Case : The slowest an algorithm can run on an input of problem size n. Eg: Insertion sort on a list of size n runs slowest when the list is sorted in the reverse order
Best Case : The fastest an algorithm can run on an input of problem size n. Eg: Insertion sort on a list of size n runs fastest when the list is sorted to start
Average Case : The average run time of an algorithm of all problem instances of size n



4.1 Asymptotic Notation

4.1.1 Table of Informal Definitions

The "Kinda like Saying" is entirely correct, it's just to wrap my head around the bounds and relations

Name	Symbol	Informal Definition	Kinda like Saying
Little Omega	ω	Lower bound	$g(n) \in \omega(f(n))$ so $g(n) > f(n)$
Big Omega	Ω	Tight Lower bound	$g(n) \in \Omega(f(n))$ so $g(n) \geq f(n)$
Big Theta	Θ	Both an upper and lower bound	$g(n) \in \Theta(f(n))$ so $g(n) = f(n)$
Big Oh	O	Tight Upper bound	$g(n) \in O(f(n))$ so $g(n) \leq f(n)$
Little Oh	o	Upper bound	$g(n) \in o(f(n))$ so $g(n) < f(n)$

4.1.2 Table of Formal Definitions

Name	Symbol	Formal Definition
Little Omega	ω	$g(n) \in \omega(f(n)) \iff \exists c > 0, n_0 > 0 \ni g(n) > cf(n) \forall n > n_0$
Big Omega	Ω	$g(n) \in \Omega(f(n)) \iff \exists c > 0, n_0 > 0 \ni g(n) \geq cf(n) \forall n > n_0$
Big Theta	Θ	$g(n) \in \Theta(f(n)) \iff \exists c_1 > 0, c_2 > 0, n_0 > 0 \ni c_1 \leq g(n) \leq c_2 f(n) \forall n > n_0$
Big Oh	O	$g(n) \in O(f(n)) \iff \exists c > 0, n_0 > 0 \ni g(n) \leq cf(n) \forall n > n_0$
Little Oh	o	$g(n) \in o(f(n)) \iff \exists c > 0, n_0 > 0 \ni g(n) < cf(n) \forall n > n_0$

4.1.3 Table of Limit Definitions

Name	Symbol	Proving with Limits
Little Omega	ω	$\lim_{n\to\infty} f(n)/g(n) = \infty$
Big Omega	Ω	$\lim_{n\to\infty} f(n)/g(n) \neq 0$
Big Theta	Θ	$\lim_{n\to\infty} f(n)/g(n) \neq 0, \infty$
Big Oh	O	$\lim_{n\to\infty} f(n)/g(n) \neq \infty$
Little Oh	o	$\lim_{n\to\infty} f(n)/g(n) = 0$

4.1.4 Properties

Transitivity :

- $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$: $f(n) \in O(h(n))$
- $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n))$: $f(n) \in \Omega(h(n))$

Reflexivity : $f(n) \in \Theta(f(n))$

Transpose Symmetry : $f(n) \in O(g(n) \iff g(n) \in \Omega(f(n))$

Symmetry : $f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n))$

Trichotomy : For any two real numbers a and b: $a > b$ or $a = b$ or $a < b$

4.2 Recurrence Relations

4.2.1 Divide and Conquer Algorithms

Three Parts:

- **Divide :** Split the problem into subproblems of the same structure
- **Conquer :** If subproblem is at the smallest possible solvable size, solve
- **Merge :** Combine the subproblem solutions into one solution

On the example Merge-Sort: Given a list A of n integers, sort the list

- **Divide :** Split list A into two lists, A_l and A_r with size $n/2$ each

- **Conquer :** Recurse until A_x has length 1 (a sorted list)
- **Merge :** Sort the sublists when merging into the larger list

Merge Sort has a runtime of:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{otherwise} \end{cases}$$

4.3 General Form

We get the general form of recurrence relations of:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(\frac{n}{b}) + D(n) + C(n) & \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathbf{a} &= \text{Number of subproblems created} \\ \mathbf{b} &= \text{What size the subproblems are divided by} \\ \mathbf{D(n)} &= \text{Time to divide} \\ \mathbf{C(n)} &= \text{Time to combine} \end{aligned}$$

4.4 Recursion Tree

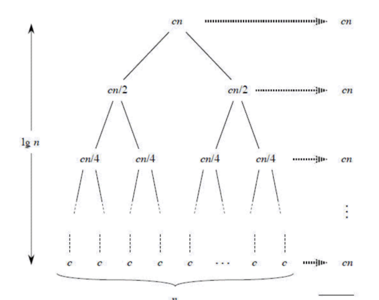
To solve a recurrence relation with a recursion tree, we build a tree and measure the cost at each height.

Recall that Merge Sort is defined as:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{otherwise} \end{cases}$$

Which builds the following tree:

Each tree has a children. Each level is divided by b in each level. In this case, each tree has 2 children, each size cut in half. There are $lg(n)$ levels. The bottom level has n subproblems. Each level has a total time of cn. This gives a general runtime of $O(nln(n))$



4.4.1 Solving

To solve a recursion tree, we need three pieces of information:

1. d : depth of the tree

2. w : width of the tree
3. c : cost at every level

Given the form of recurrence relations:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

Then we get the following summation where $d = \log_b(n)$, $w = a^k$, $c = w * f(\frac{n}{b^k})$:

$$T(n) = \sum_{k=0}^{d} c$$
$$T(n) = \sum_{k=0}^{\log_b(n)} (a^k * f(\frac{n}{b^k}))$$

4.5 Substitution

To solve a recurrence relation with substitution, we guess a bound and use mathematical induction to prove it's correctness

1. Guess the general form of the solution
   - For mergesort, $T(n) \in O(nlg(n))$
2. Use induction to show it works for some constants
   - Assume $T(n') \le cn'lg(n') \forall n' < n$
   - Show $T(n) \le cnlgn$

$$\begin{aligned} T(n) &= 2T(\lfloor \frac{n}{2} \rfloor) + n \\ &\le 2(c\lfloor \frac{n}{2} \rfloor lg(\lfloor \frac{n}{2} \rfloor)) + n \\ &\le cnlg(\lfloor \frac{n}{2} \rfloor) + n \\ &\le cnlg(n) - cnlog(2) + n \\ &\le cnlg(n) - cn + n \\ &\le cnlg(n) \end{aligned}$$

4.6 Master Theorem

To solve a recurrence relation with the master theorem, we directly provide bounds for recurrence.

Given that $a \geq 1$ and $b \geq 1$ and are constants, let $f(n)$ be a function and let $T(n)$ be defined on non-negetive integers with the following recurrence:

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

Note: We interpret $\frac{n}{b}$ *to be either* $\lfloor \frac{n}{b} \rfloor$ *or* $\lceil \frac{n}{b} \rceil$

$T(n)$ has the following bounds:

- When $f(n) \in O(n^{\log_b(a-\varepsilon)})$ for some constant ε
  - $T(n) \in \Theta(n^{\log_b(a)})$
- When $f(n) \in \Theta(n^{\log_b(a)})$
  - $T(n) \in \Theta(n^{\log_b(a)} \log(n))$ *Note: What base?*
- When $f(n) \in \Omega(n^{\log_b(a+\varepsilon)})$ for some constant ε, and $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n
  - $T(n) \in \Theta(f(n))$

4.6.1 Replacement of Variables

Page 86 of CLRS

The recitation example is:

$$T(n) = \sqrt{n}T(\sqrt{n}) + n$$

This doesn't follow the master theorem exactly, but we can do a replacement of variables

First, we state $m = \log(n)$:

$$\begin{aligned} m &= \log(n) \\ n &= 2^m \\ \sqrt{n} &= 2^{m/2} \end{aligned}$$

We now we solve:

$$
\begin{aligned}
T(2^m) &= 2^{m/2}T(2^{m/2}) + 2^m && \text{Replace variables} \\
\frac{T(2^m)}{2^m} &= \frac{2^{m/2}T(2^{m/2})}{2^m} + \frac{2^m}{2^m} && \text{Divide both sides by } 2^m \\
\frac{T(2^m)}{2^m} &= \frac{T(2^{m/2})}{2^{m/2}} + 1 && \text{Simplify} \\
S(m) &= \frac{T(2^m)}{2^m} && \text{Define } S(m) \\
S(m) &= S(m/2) + 1 && \text{Replace functions}
\end{aligned}
$$

Now we can apply the master theorem to $S(m)$ where $a = 1$, $b = 2$ and $f(m) = 1$:

$$
\begin{aligned}
f(m) &\in \Theta(m^{\log_b(a)}) \\
1 &\in \Theta(m^{\log_2(1)}) \\
1 &\in \Theta(m^0) \\
1 &\in \Theta(1)
\end{aligned}
$$

This means $S(m) \in \Theta(\log(m))$. We then just put $T(n)$ back in:

$$
\begin{aligned}
S(m) &= \frac{T(2^m)}{2^m} && \text{Previously defined} \\
T(2^m) &= 2^m S(m) \\
T(2^m) &\in \Theta(2^m \log(m)) \\
T(n) &\in \Theta(n \log(\log(n))) && \text{Replace } n
\end{aligned}
$$

5 Expected Runtime - Randomization

Indicator Variable : $\mathbb{I}\{E\}$ is 1 when event E happens, 0 otherwise

Expectation : $\mathbb{E}[x]$ is the probability that x is true. Can also be used as $\mathbb{E}[f(n)]$ as the expected runtime of $f(n)$

For example, given an array A with n elements, r is an element chosen uniformly at random. Define indicator variable E_i as:

$$E_i = \mathbb{I}\{A[i] == r\}$$

Given array A and $r = 5$:

$$\begin{array}{cccccc} A & [& 2 & 5 & 6 & 11 &] \\ E_i & [& 0 & 1 & 0 & 0 &] \end{array}$$

This means $\mathbb{E}[E_i] = \frac{1}{n}$

5.1 Quick Sort Example

Quick sort, but instead of choosing the element in the middle of the array as the pivot, choose a random r.

- **Worst Case :** $O(n^2)$ when the random r chosen is always the largest or smallest element in an array
- **Expected Case :** $O(n)$

Worst case run time is:

$$T(n) = T(max(r-1, n-r)) + O(n)$$

Since the array splits into two parts based on r but only one is travelled. $O(n)$ is the time to split the array. The max function is based on r and the values being compared are:

r	1	2	...	$n-1$	n
$r-1$	0	1	...	$n-2$	$n-1$
$n-r$	$n-1$	$n-2$	...	1	0

This means max can be also defined as:

$$max(r-1, n-r) = \begin{cases} r-1 & r > \lceil \frac{n}{2} \rceil \\ n-r & r \leq \lceil \frac{n}{2} \rceil \end{cases}$$

The define our indicator variable:

$$X_r = \mathbb{I}\{\text{Subarray } A[p...q] \text{ has exactly } r \text{ elements}\}$$

It's expectation:

$$\mathbb{E}[X_r] = \frac{1}{n}$$

Then put it all together to find $\mathbb{E}[T(n)]$:

$$\begin{aligned}
\mathbb{E}[T(n)] &\leq \mathbb{E}[\sum_{r=1}^{n} X_r T(max(r-1, n-r)) + O(n)] \\
&= \sum_{r=1}^{n} \mathbb{E}[X_r T(max(r-1, n-r)) + O(n)] \\
&= \sum_{r=1}^{n} \mathbb{E}[X_r]\mathbb{E}[T(max(r-1, n-r))] + \mathbb{E}[O(n)] \\
&= \sum_{r=1}^{n} \frac{1}{n}\mathbb{E}[T(max(r-1, n-r))] + O(n) \\
&= \frac{1}{n}\sum_{r=1}^{n} \mathbb{E}[T(max(r-1, n-r))] + O(n) \\
&= \frac{2}{n}\sum_{r=\lfloor \frac{n}{2} \rfloor}^{n-1} \mathbb{E}[T(k)] + O(n) \\
\mathbb{E}[T(n)] &\leq \frac{2}{n}\sum_{r=\lfloor \frac{n}{2} \rfloor}^{n-1} \mathbb{E}[T(k)] + O(n) \\
\mathbb{E}[T(n')] &\in O(n') && \text{Assume} \\
\mathbb{E}[T(n')] &\leq cn' \\
\mathbb{E}[T(n)] &\leq \frac{2}{n}\sum_{r=\lfloor \frac{n}{2} \rfloor}^{n-1} cn + an && \text{Substitute} \\
\mathbb{E}[T(n)] &\leq \frac{2c}{n}(\frac{n^2-n}{2} - \frac{\frac{n^2}{4} - \frac{3n}{2} + 2}{2}) + an \\
\mathbb{E}[T(n)] &\leq cn - (\frac{cn}{4} - \frac{c}{2} - an) \leq cn && \text{Induction Holds} \\
\mathbb{E}[T(n)] &\in O(n) && c > 4a, n \geq \frac{2c}{c-4a}
\end{aligned}$$

5.2 Probabilistic Analysis

Deterministic : Calculated. Not random

Probabilistic Anaylsis : Take the average performance across the distribution of random inputs leading to average case running time

Random Algorithms : Make random decisions in algorithms and take the expectation of running times across all decisions leading to an expected running time.

Probablistic analysis and randomized algorithms have three main benefits

1. Randomized algorithms can have better expected running time or performances

2. Deterministic algorithms can be much harder to analyze and have larger constants
3. Analyzing worst case for problems with random input can be too pessimistic

5.3 Linearity of Expectation

Calculate the expected number of head when flipping 4 coins:

Exhaustive Checking

Use exhaustive checking and axioms of probability, 4 * Pr(4 heads) + 3 * Pr(3 heads) + 2 * Pr(2 heads) + 1 * Pr(1 head) + 0 * Pr(0 head)

$$(4 * \binom{4}{4} * \frac{1}{2^4}) + (3 * \binom{4}{3} * \frac{1}{2^4}) + (2 * \binom{4}{2} * \frac{1}{2^4}) + (1 * \binom{4}{1} * \frac{1}{2^4}) =$$
$$\frac{4}{16} + \frac{12}{16} + \frac{12}{16} + \frac{4}{16} =$$
$$2$$

Indicator Random Variable

$X_k = \mathbb{I}\{\text{Coin } k \text{ is heads}\}$

Number of heads is the sum of the indicator variables.

$$\mathbb{E}[\sum_{k=1}^{4} X_k]$$
$$\sum_{k=1}^{4} \mathbb{E}[X_k]$$
$$\sum_{k=1}^{4} \frac{1}{2}$$
$$4\frac{1}{2}$$
$$2$$

6 Hashing

Assuming we had a set K with keys drawn from the universe $U = \{0, 1, ..., m\}$ the following table is a table of runtimes of functions on set K and key k:

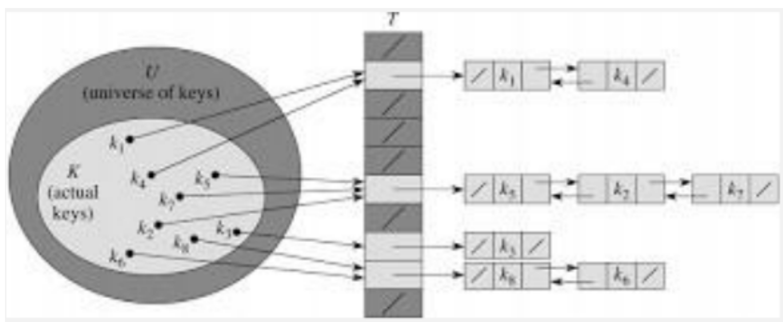
	Stack	**Queue**	**Linked List**
$Insert(K, k)$	$O(1)$	$O(1)$	$O(n)$ (Sorted)
$Search(K, k)$	$O(n)$	$O(n)$	$O(n)$
$Delete(K, k)$	$O(n)$	$O(n)$	$O(n)$

Hash Table : A data structure that addresses elements based on a hash function

Hash Function : A deterministic constant time operation to compute an index in the hash table for a key k

Collision : When two keys are mapped to the same spot

Chaining : Using linked lists to put multiple keys in the same bucket



The fastest hash table would be one with no collisions, which would require a hash table with size $|U|$. If $|K|$ is small, memory is wasted. How many insertions are required before we can expect a collision?

$$X_{ij} = \mathbb{I}\{i \text{ and } j \text{ hash to the same location}\}$$

Then if we use X to represent the number of pairs with the same hash, we can build and solve the following summation:

$$
\begin{aligned}
\mathbb{E}[X] &= \mathbb{E}[\sum_{i=1}^{k} \sum j = i + 1^k X_{ij}] \\
\mathbb{E}[X] &= \sum_{i=1}^{k} \sum j = i + 1^k \mathbb{E}[X_{ij}] \\
\mathbb{E}[X] &= \binom{k}{2} \mathbb{E}[X_{ij}] \\
\mathbb{E}[X] &= \binom{k}{2} \frac{1}{m} \\
\mathbb{E}[X] &= \frac{k(k-1)}{2m}
\end{aligned}
$$

What if we have a hash table of length m with n elements. What is the expected length of bucket b?

$$
\begin{aligned}
X_j &= \mathbb{I}\{j\text{th element hashes to } b\} \\
\mathbb{E}[X_j] &= \mathbb{E}[\sum_{j=1}^{n} X_j] \\
\mathbb{E}[X_j] &= \sum_{j=1}^{n} \mathbb{E}[X_j] \\
\mathbb{E}[X_j] &= \sum_{j=1}^{n} \frac{1}{m} \\
\mathbb{E}[X_j] &= \frac{n}{m}
\end{aligned}
$$

This basically says when $n > m$, then expect to find $\frac{n}{m}$ elements in each bucket, which is like the pidgeon hole principle.

How many items do we need to insert to be sure we fill all positions in the hash table?

$$
\begin{aligned}
X_i &= \mathbb{I}\{\text{New hash table entry is used after using } i - 1 \text{ entries}\} \\
\mathbb{E}[X_i] &= \frac{m - 1 + 1}{m} \\
Y_i &= \text{Number of inserts to use the } i\text{th distinct entry} \\
\mathbb{E}[Y_i] &= \frac{1}{\mathbb{E}[X_i]}
\end{aligned}
$$

Then to find the total number of insertions to fill m buckets:

$$\mathbb{E}[T(m)] = \mathbb{E}[\sum_{i=1}^{m} Y_i]$$
$$\mathbb{E}[T(m)] = \sum_{i=1}^{m} \mathbb{E}[Y_i]$$
$$\mathbb{E}[T(m)] = \sum_{i=1}^{m} \frac{m}{m - i + 1}$$
$$\mathbb{E}[T(m)] = \Theta(m \log(m))$$

7 Binary Trees

Tree : A data structure with heirarchy

Binary Tree : A tree with two children, each being a distinct left or right child.

Root : The "highest" node. Has no parents

Leaf : The "deepest" node. Has no children

Height : The number of children from a node to a leaf

Depth : The number of parents from a node to the root

Convex Function : A function $f(x)$ is convex if:

$$\forall x, y \forall 0 \le \lambda \le 1 : f(\lambda x + (1-\lambda)y) \le \lambda f(x) + (1-\lambda)f(y)$$

Jensen's Inequality : When we apply a convex function $f(x)$ to a random variable X, then:

$$\mathbb{E}f(X) \ge f(\mathbb{E}X)$$

Complete Rooted Binary Tree : A binary tree where all leaves have the same depth and all internal nodes have degree 2

Full Rooted Binary Tree : A binary tree where all nodes are either leaves or have a degree of exactly 2

Note: Appendix B of CLRS has lots of info

7.1 Search Trees

A specific type of binary tree where the left child n_l is less than node n which is less than the right child n_r.

```
struct Node
key // The value stored in the node
left // Pointer to left child
right // Pointer to right child
p // Pointer to parent
function Search(x, T)
    // O(h)
    r := T.root
    if r == NIL then return 0
    end if
    if x == r.key then return 1
    else if x > r.key then
        Search(x, r.right)
    else if x < r.key then
        Search(x, r.left)
    end if
end function
```

Given n random elements in any order placed randomly into the tree, the height grows:

Worst Case : $O(n)$: Each element is the same child. All left children or all right children. Height grows linearly.

Best Case : $O(\log(n)$: There are an equal number of left and right children placed such that a height is filled before moving onto the next.

Expected Case :

We define the height of node n based on it's children as:

$$h(n) = max(h(n_l), h(n_r)) + 1$$

Given a binary tree with n nodes, the height is X_n. The exponential height is $Y_n = 2^{X_n}$

Suppose there are $i-1$ nodes in the left child. This means the exponential height of the tree can be defined as:

$$Y_n = 2 * max(Y_{i-1}, T_{n-i})$$

By Jensen's Inequality, if we prove $\mathbb{E}Y_n$ is polynomial in n, then $\mathbb{E}X_n$ is logarithmic in n. To do this we need an indicator variable $Z_{n,i}$

$$Z_{n,i} = \mathbb{I}\{\text{Root is the } i\text{th order statistic of the set of } n \text{ ints}\}$$

Now plug and chug:

$$\begin{aligned}
Y_n &= \sum_{i=1}^{n} Z_{n,i}(2 * max(Y_{i-1}, Y_{n-i})) \\
\mathbb{E}Y_n &= \mathbb{E}\sum_{i=1}^{n} Z_{n,i}(2 * max(Y_{i-1}, Y_{n-i})) \\
&= \sum_{i=1}^{n} \mathbb{E}Z_{n,i}(2 * max(Y_{i-1}, Y_{n-i})) && \text{Linearity of Expectation} \\
&= \sum_{i=1}^{n} \mathbb{E}Z_{n,i}\mathbb{E}(2 * max(Y_{i-1}, Y_{n-i})) && \text{Independence} \\
&= \sum_{i=1}^{n} \frac{1}{n}\mathbb{E}(2 * max(Y_{i-1}, Y_{n-i})) \\
&= \frac{2}{n}\sum_{i=1}^{n} \mathbb{E}max(Y_{i-1}, Y_{n-i}) \\
&\leq \frac{2}{n}\sum_{i=1}^{n} \mathbb{E}Y_{i-1} + \mathbb{E}Y_{n-i}
\end{aligned}$$

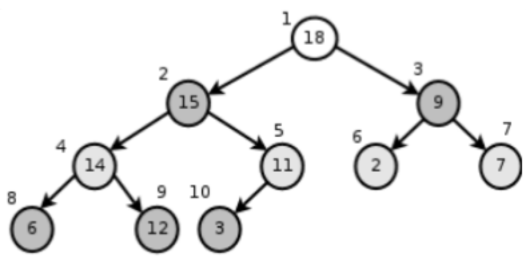
Solve the recurrence relation with substitution where we guess $\mathbb{E}Y_n \in O(n^3)$ and show that $\mathbb{E}Y_n$ is polynomial meaning a randomly built binary search tree will have an expected height $O(\log(n))$

7.2 Heaps

Combination of notes from class and CS-260 with Kurt (*sexy heap shape*)

A binary tree with two extra properties:

1. It's full. It fills left to right
2. Children are not X than parent
   - **Min Heap :** Minimum element is at the top of the heap
   - **Max Heap :** Maximum element is at the top of the heap



Can be implemented easily over an array *Note: Indexed at 1*:

18	15	9	14	11	2	7	6	12	3
1	2	3	4	5	6	7	8	9	10

---

function PARENT(i)
    return $\lfloor \frac{i}{2} \rfloor$
end function

function LEFT(i)
    return $2i$
end function

function RIGHT(i)
    return $2i + 1$
end function

---

function UPHEAP(i, H)
    // From CS260 with Kurt
    // $O(\log(n))$
    // For a min heap, replace the $>$ with a $<$
    while $i > 1$ and $H[i] > H[\text{Parent}(i)]$ **do**
        swap(i, Parent(i), H)

```
        i := Parent(i)
    end while
end function

function Downheap(i, H)
    // From CS260 with Kurt
    // Vasilis calls this MaxHeapify
    // O(log (n))
    if Left(i) ¿ H.size then return
    end if
    li := Left(i) // The index of the larger child
    // For a min heap, replace the < with a >
    if Right(i) ≤ H.size and H[li] < H[Right(i)] then
        li := Right(i)
    end if
    // For a min heap, replace the > with a <
    if H[i] < H[li] then
        swap(i, li, H)
        Downheap(li, H)
    end if
end function
```

```
function Insert(x, H)
    // From CS260 with Kurt
    // O(log (n))
    ResizeArray?()
    H.size := H.size + 1
    H[H.size] := x
    Upheap(H.size, H)
end function

function Remove(H)
    // From CS260 with Kurt
    // O(log (n))
    if Empty?() then return
    end if
    ResizeArray?()
    rv := H[1]
    H[1] := H[H.size]
    H.size := H.size - 1
    Downheap(1, H) return rv
end function

function BuildMaxHeap(A)
    // From CS457 with Vasilis
    // O(n log (n))
    A.size := A.length
    for i := ⌊A.length / 1⌋ down to 1 do
        Downheap(i, A)
    end for
end function

function Heapsort(A)
```

```
    // From CS457 with Vasilis
    // O(n log(n))
    BuildMaxHeap(A)
    for i := A.length down to 2 do
        swap(1, i, A)
        A.size := A.size - 1
        Downheap(1, A)
    end for
end function
```

7.3 RedBlack Trees

A balanced binary search tree. So their height is $O(\log(n))$. It also has the following color properties:

- Every node is **red** or **black**
- **Root** is **black**
- **Leaves** are **black**
- If a **node** is **red**, both of **its children** are **black**

This means that all paths from a node x to a **leaf** have the same number of **black nodes**.

```
function RB-Insert(T, z)
    y := T.nil
    x := T.root
    while x ≠ T.nil do
        x := y
        if z.key < x.key then
            x := x.left
        else
            x := x.right
        end if
    end while
    z.p := y
    if y == T.nil then
        T.root := z
    else if z.key < y.key then
        y.left := z
    else
        y.right := z
    end if
    z.left := T.nil
    z.right := T.nil
    z.color := RED
    RB-Insert-Fixup(T, z)
end function
```

```
function RB-Insert-Fixup(T, z)
    while z.p.color == RED do
        if z.p == z.p.p.left then
            y := z.p.p.right
            if y.color == RED then
                z.p.color := BLACK
                y.color := BLACK
                z.p.p.color := RED
                z := z.p.p
            else
                if z == z.p.right then
                    z := z.p
                    Left-Rotate(T, z)
                end if
                z.p.color := BLACK
                z.p.p.color := RED
                Right-Rotate(T, z.p.p)
            end if
        else
            // Same as then clause with "left" and "right" exchanged
        end if
    end while
    T.root.color := BLACK
end function
```

7.4 Structural Induction

Regular induction has two main parts:

1. **Base Case :** Prove it works for the smallest case
2. **Inductive Hypothesis :** Assume it works for the +1 case. Show our assumption was true

Typically this is used with arrays, where the base case is an array of no elements or one element, then the inductive hypothesis is based around what happens when an element is added to the array.

With trees, it's not explicitly clear what the induction variable is since it could be the number of nodes, or edges, or the height. The proof instead follows the structure of a recursive definition since the children of trees are trees.

1. The claim holds for trees of a single node
2. If the claim holds for trees A and B, then the claim holds for a new tree consisting of a root with children A and B

8 In Class Examples

8.1 L01 : Maximum Subarray Problem

Lecture 01 : 9/20/2021

Problem :

Given an array A of n numbers, both positive and negative, find a contiguous subarray with the maximum sum of numbers.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

In the above array, the maximum subarray is $[8, 11]$

Questions :

- What is the first, simple, algorithm that comes to mind?
- What is the running time of this algorithm?
- Can you come up with a divide and conquer algorithm?

Observations :

Brute Force :

- Find all subarrays, calculate sum, compare
- $\sum_1^n 1$ subarrays
  - 16 + 15 + ... + 2 + 1 subarrays
  - $\approx n^2$ subarrays
- Computing sum takes $\approx n$
- $\in O(n^3)$

Optimizations :

- Cannot be less than linear
  - Needs to be $\in O(n)$ to read all input
- Can we get to $\in O(n^2)$
  - To get to $\in O(n^2)$ would we need to use more space?
- My partial solution
  - Break the array into parts based on sign
  - Find sum of adjacent elements of same sign $\in O(n)$

- $A = [13, 5, -22, -3, 5, -6, -2]$ becomes $B = [18, -25, 5, -8]$
- Need another array C to store indices of subarrays
- Find the largest element of the array $\in O(n)$
- Store start and end indices of largest element $B[m]$
- If $B[m] < 0$ return
- If $B[m] + B[m+1] + B[m+2] > B[m]$ increase size of maximal array
- If $B[m] + B[m-1] + B[m-2] > B[m]$ increase size of maximal array
- Repeat until niether side can be increased more $\in O(n)$?

9 Dynamic Programming

Dynamic programming is recursion, but we remember things along the way. If we find ourselves recalculating something many times or there are overlapping subproblems, remember their results. The fibonacci example is a good example of this.

function FIBONACCI(n)
    if $n == 0$ or $n == 1$ **then** Return n
    elsereturn Fibonacci($n-1$) + Fibonacci($n-2$)
    end if
end function

$$T(n) = T(n-1) + T(n-2) + 1$$

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

Insert tree image

Instead we can approach the problem as:

m := An array of length n filled with only 0s
function FIBONACCI(n)
    if $n == 0$ or $n == 1$ **then** Return n
    else if $m[n]$!= 0 **then return** $m[n]$
    else$m[n]$:= Fibonacci($n-1$) + Fibonacci($n-2$)
return $m[n]$
    end if
end function

Insert tree image

10 Graphs

Given a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, then:

<u>**General Vocab**</u>

Vertex : A "node" of the graph, $|V| = n$

Edge : Connects two vertices, $|E| = m \in O(n^2)$

Path :

Cycle :

Degree of a Vertex :

Maximum/Minimum Degree :

Maximum Number of Edges :

Connected Components :

Shortest Weighted Path :

Shorted Unweighted Path :

Distance of Two Vertices :

Tree :

Spanning Tree :

<u>**Types of Graphs**</u>

Sub-Graph : A subset of vertices and edges in a graph

Connected Graph : For every pair of vertices (u, v) in G, there is a path connecting them

Weighted Graph : Each edge has an associated weight or cost when traveling

Undirected Graph : Edge $(u, v) = (v, u)$

Directed Graph : Edges $(u, v) \neq (v, u)$

Acyclic Graph : A graph without cycles

Bipartite Graph :