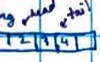
Simple Data Structures

Array: continuous set of containers
- Goal: random access in constant time
- e.g. int number[5];
- Access time: $\Theta(1)$ (constant time)
- Insert time: Dependent on where you want to insert
  - constant time $\Theta(1)$ if space available STRENGTH
  - else, allocate new array, copy all elems, insert new elem
    - weakness, requires continuous memory

Vector: STL class, a resizeable array
- e.g.
```
Vector<int> v1;
v1.push_back(1);
v1.push_back(2);
```
- CONTINUOUS MEMORY (rand access in const time)
- To add: double size for each element inserted to a full vector
  - Efficiency: time to insert n elements to an empty vec?
    $T(n) = T(\frac{n}{2}) + \frac{n}{2} \Rightarrow \Theta(n)$ (MM case 3)
    $\Theta(n)$ amortized time to insert 1 elem;
    WORST CASE: one insert expensive, next few cheap
  - For deleting, keep a separate threshold for halfing size
- ITERATOR: Object that points to an element in a range (vec, list)
```
e.g. vector<int>::iterator iter = v1.begin
For(; iter != v1.end(); iter++)
{
    cout << *iter; //or v1[i], since vector is continuous
}
```

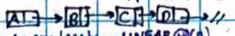
Queue: FIFO
- enqueue: insert elem to end
- dequeue: remove from beginning
- Implement with an array
- to combat shifting, loop back w/ mod operation



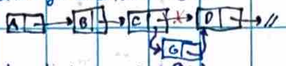
Stack: LIFO
- push: add to tail
- push: remove from tail
- implement w/ array: just need tail pointer

Deck: Double-ended queue STL class
- does not use continuous memory
- can insert efficiently to end & beginning (in CONSTANT TIME)
- Implement w/ linked list: (each node element has a payload (key) and a pointer to the next node)

e.g.



- Insertion to any location: LINEAR $\Theta(n)$



- Delete at any location: LINEAR $\Theta(n)$
- Double linked-list - each node also points to previous node, allows insert/delete to tail in CONSTANT time

Non-Comparison-Based Sorting

~~Comparison-Based~~

Counting Sort
- Assumption: n inputs, each in range $[0, m)$, all $\in \mathbb{Z}$
- e.g. 3 5 4 1 3 4 1 4
  - create array B of size m init to 0
  - for each occurence of k in A, B[k]++

```
B: 0 1 2 3 4 5
   0 0 0 0 0 0
     1   1 1 1
     2   2 2
           3
```

- Spit out index i of B, B[i] times:
  1 1 3 3 4 4 4 5
  - Faster than n log n b/c no comparisons
- RUNTIME? $\Theta(n+m)$ (equal to max(n, m))
- SPACE? If stable, $\Theta(m+n)$
- Faster, but not versatile

Radix Sort
- Assumption: n inputs, all k digits long, each digit in range $[0, d)$ ← base
- Input A, k
```
For i=1 to k
    use a stable sort (i.e. counting sort) to sort array on digit i.
```
- Runtime: $\Theta(k(n+d))$ ← k times counting sort
- Space: $\Theta(n)$ → one extra array (START W/ LAST DIGIT)
- can be used for any base, chars, etc

SORTING ALGORITHMS

Comparison Based

Bubble Sort: Repeatedly pass through the array, swap adjacent elements that are out of order
- Data Structure: Array
- Runtime: $\Theta(n^2)$ - worst case
  $\Theta(n^2)$ - best case
  $\Theta(n)$ - best case w/ code mod
- ★ After first iteration, largest element is at the end ★

```
For(i=0; i<|A|-1)
{
    swapped = False
    For(j=0 to |A|-2)
    {
        if A[j]>A[j+1]
        {
            swap A[j], A[j+1]
            Swapped = TRUE
        }
    }
    if(!Swapped) DONE
}
```

ONE ITERATION
```
8 4 6 9 2 3  SWAP
4 8 6 9 2 3  SWAP
4 6 8 9 2 3  DO NOTHING
4 6 8 9 2 3  SWAP
4 6 8 2 9 3  SWAP
4 6 8 2 3 9  Done w/ first iteration
```

Advantages:
- space
- efficient for small inputs, almost sorted
- quickly finds max
- Simple & easily parallizeable

Disadvantages:
- inefficient for large inputs → non small

Selection Sort: find min, swap w/ beginning, repeat → i, min
- Runtime: $\Theta(n^2)$ in every case
- Space: sort in place (constant)
```
For i=0 to |A|-2
    min=i
    For j=i+1 to |A|-1
        if A[j]<A[min]
            min=j
    Swap A[i], A[min]
```

e.g.
```
8 4 6 9 2 3  SWAP
2 4 6 9 8 3  SWAP
2 3 6 9 8 4  SWAP
2 3 4 9 8 6  SWAP
2 3 4 6 8 9  NO SWAP
             DONE
```

Advantages:
- Find min quickly (good/bad)
- simple, but more complex than bubble sort
- consistent runtime → for security

Disadvantages:
- inefficient for non small inputs
- best case is bad

Insertion Sort: place each value in its correct location as you reach it, slide everything else over
- runtime: worst case $\Theta(n^2)$
  best case $\Theta(n)$
- Space: sort in place (constant)

```
for i=0 to |A|-1
    value = A[i]
    j=i-1
    while j≥0 and A[j] > value
        A[j+1] = A[j]
        A--
    A[j+1] = value
```

Advantages:
- constant space
- Faster for best case

Disadvantages
- slow for non-small inputs
- ~~Worse for best case~~
- more complex than bubble selection

Merge Sort: Divide & Conquer Algorithm
```
[ MergeSort(A[1... ⌊n/2⌋])
[ MergeSort(A[⌊n/2⌋+1...n])
[ merge halves
```
- Runtime: $T(n) = 2T(\frac{n}{2}) + n$
  $\Theta(n \log n)$ (All cases)
- Space: $\sim\Theta(n)$

e.g.
```
7 2 9 4 | 3 8 6 1
7 2 | 9 4    3 8 | 6 1
7|2   9|4   3|8   6|1
7 2   9 4   3 8   6 1
2 7   4 9   3 8   1 6
2 4 7 9     1 3 6 8
1 2 3 4 6 7 8 9  DONE
```

Advantages
- Faster!

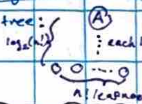
Disadvantages
- not constant space
- bad for smalls

Quicksort: Randomized divide & conquer
1. Select a pivot at random, or rightmost element
2. Place all elements ≤ pivot to left of pivot
3. Place all elements > pivot right of pivot
4. Quicksort (left of pivot partition)
5. Quicksort (right of pivot partition)
- Space: could be done w/ no extra space (const)
- Runtime: $\Theta(n \log n)$ avg & best
  $\Theta(n^2)$ WORST CASE (piv max or min)

```
10 15 2 1 12 3
2 1 3 10 12 15
1 2 3 10 12 15
1 2 3 10 12 15
```

Information Theoretic Lower Bound on Comparison-Based Sorting Algorithm Runtimes: n log n

Decision tree:



Bound this?
STERLING:
$\log(n!) \doteq \log(\frac{n}{2})^{n/2}$
Runtime is $\Omega(n \log n)$

[n! possible orderings of n numbers, tree height = lg n!
→ by Sterling's approx, $\Theta(n \log n)$ or we missed some sequence!]

e.g. Provide & solve the recurrence for the following functions:
```
int c(int n) {
    if(n==n/2)
        return 1;
    else
        return 12 * c(n/120)
}
```
$T(n) = T(\frac{n}{120}) + c \Rightarrow$ case 2 MM,
$T(n) \in \Theta(\log n)$

```
int b(int n) {
    if (n==0) return 1;
    else return 2*B(n-1)
}
```
$T(n) = T(n-1) + c \Rightarrow T(n)$ is $\Theta(n)$

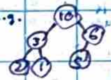
Bucket Sort
- Assumption: input size n, uniformly distributed $[0, 1)$ ← # of buckets ← or anything
- bucketSort(A)
```
For i=0 to n-1
    insert A[i] into bucket B[⌊n A[i]⌋]
For i=0 to n-1
    Sort bucket B[i] w/ insertion sort
concatenate buckets B[0], B[1], ... B[n-1]
```
- Bucket sort can be adapted to other ranges (A-Z, [0-1]?)
- e.g. 7.2, 1, 3.2, 7.3, 3.1, 7.1 BUCKET SORT? ✓ ∈ [0, 8)

```
0.  1.  2.  3.   4.  5.  6.  7.
    1       3.2              7.2
            3.1              7.3
                             7.1
1 3.1 3.2 7.1 7.2 7.3
```

- Runtime: $\Theta(n)$ on avg
  $\Theta(n^2)$ worst case
- Space: $\Theta(n)$

STABLE SORT: identical elements keep original order (extend array B w/ links to make counting sort stable)

Heap Sort
- heap: a complete binary tree w/ max-heap or min-heap property ← all levels full except bottom
- e.g.

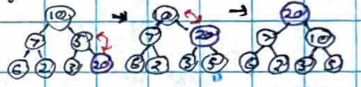


0	1	2	3	4	5	6
X	10	3	6	2	1	5

← heap can be stored in an array!
children of i: 2i, 2i+1
Parent of i: ⌊i/2⌋

- FindMax of Max Heap: Go to first element in array, $\Theta(1)$

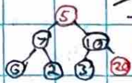
e.g. Insert 20



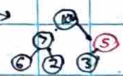
- insert (max heap)
```
heap_insert(A, key)
    heap_size(A)++
    i <- heap_size(A)
    A[i] <- key
    while(i>1 and A[parent(i)] < A[i])
        swap(A[i], A[parent(i)])
        // bubble up
        i <- parent i
```

- RUNTIME: best case $\Theta(1)$, worst case $\Theta(\log n)$

- EXTRACT MAX: swap last item w/ root, delete last item, "heapify" (bubble down)
```
Heapify(A, i)  // i is new violating node
    if A[i] ≥ max(left(i), right(i))
        do nothing
    else
        swap(A[i], max_child)
        Heapify(A, i of max child)
```



- delete 20, compare A[1] to children
- SWAP RIGHT



- Compare A[3] to children
- DONE

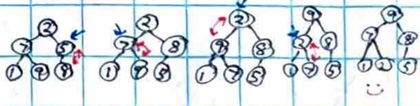
- Runtime? Heapify $\Theta(\log n)$ worst case
  Extract max $\Theta(\log n)$ worst case (heapify + const runtime)

- Sorting with a heap:
  - insert n elements & extract max n times
    OR
  - build a heap from a random array ($\Theta(n)$) and extract max n times
- RUNTIME: $\Theta(n \log n)$ both cases

- Increase key: increase priority (val) of an element
  - increase val of i, heapify
    OR
  - delete & reinsert
- RUNTIME: $\Theta(\log n)$ either way (worst case)

- Delete: Swap w/ last element, delete, heapify/bubble up
  - Runtime: $\Theta(\log n)$

- Build a Heap From a Random Array 2 7 5 1 9 8
  - Heapify from right ⌊|A|/2⌋ to left (skip leaves)
  - Runtime: $\sum_{i=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{i+1}} \rceil \cdot i \approx n/2 \sum_{i=0}^{\lg n} \frac{i}{2^i} \le \frac{n}{2} \sum_{i=0}^{\infty} \frac{i}{2^i}$
    $\le \frac{n}{2} \cdot \frac{1}{2} / (1-(1/2))^2$
    WORST CASE → $\le \Theta(n)$



Domain-Range

$T(n) = T(\sqrt{n}) + c_1\sqrt{n} + c_2$

Let $n = 2^m$

$T(2^m) = T(2^{m/2}) + 2^{m/2}$

Let $S(m) = T(2^m)$. Then:

$S(m) = S(\frac{m}{2}) + 2^{m/2}$

Master thm: Case 3 $a=1, b=2, f(n) = \Theta(2^{m/2})$

$S(m) = \Theta(2^{m/2}) \xrightarrow{\text{revert}} T(n) = \Theta(\sqrt{n})$

Sadie Allen
U02945217

Summations

- **Sum of Squares:** $\sum_{k=1}^{n} k^2 = \frac{n(n-1)(2n+1)}{6}$ $\qquad \frac{(n)(n+1)}{2}$
- **Sum of Cubes:** $\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4}$
- **Arithmetic Series:** $\sum_{k=1}^{n} k = 1+2+3+\dots n = \frac{1}{2}n(n+1)$
  - diff. & mult. by x: $\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$
- **Geometric Series:** for real $x \neq 1$, $\sum_{k=0}^{n} x^k = 1 + x + x^2 + \dots x^n = \frac{x^{n+1}-1}{x-1}$
  - infinite decreasing geometric series: $|x|<1$, $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$
- **Harmonic Series:** nth harmonic # $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k} = \ln(n) + \text{const}$
- **Telescoping Series:** $\sum_{k=1}^{n}(a_k - a_{k-1}) = a_n - a_0$, $\sum_{k=0}^{n-1}(a_k - a_{k+1}) = a_0 - a_n$

  $\sum_{k=1}^{n-1} \frac{1}{k(k+1)} = \sum_{k=1}^{n-1}\left(\frac{1}{k} - \frac{1}{k+1}\right) = 1 - \frac{1}{n}$
- **Bounding Summations:** $\sum_{k=1}^{n} a_k \le n * a_{max}$, $\sum_{k=1}^{n} a_k \ge n * a_{min}$

  e.g. $n * 1 \le \sum_{k=1}^{n} k \le n * n = n^2$

  ratio of 2 consec. elements
- **Bounding by a Geometric Series:** Given $\sum_{k=0}^{n} a_k$, where $\frac{a_{k+1}}{a_k} \le r$ $\forall k \ge 0$, $0<r<1$, then

  $\sum_{k=0}^{n} a_k \le \sum_{k=0}^{\infty} a_0 r^k = a_0 \sum_{k=0}^{\infty} r^k = a_0\left(\frac{1}{1-r}\right)$

  e.g. Find a tight upper bound

  $\sum_{k=1}^{\infty} \frac{k}{3^k} = \sum_{k=0}^{\infty} \frac{k+1}{3^{k+1}} \le \frac{a_0}{1-r}$ $\quad a_0 \to 1/3$ $\quad r = \frac{\frac{k+2}{3^{k+2}}}{\frac{k+1}{3^{k+1}}} = \frac{1}{3}\left(\frac{k+2}{k+1}\right) \le \frac{2}{3}$

  $\le \frac{1}{3} \cdot \frac{1}{1-2/3} = 1$
- **Bound By Integrals:** $\int_{m-1}^{n} f(x)\,dx \le \sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)\,dx$

  e.g. Find an upper bound: $\sum_{k=1}^{n} k^4 \le \int_1^{n+1} x^4\,dx = \frac{x^5}{5}\Big]_1^{n+1} = \frac{(n+1)^5 - 1}{5}$
- **Splitting Summations:** (To find a tight lower bound)

  e.g. $\sum_{k=1}^{n} k = \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^{n} k \ge \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n} n/2 = \left(\frac{n}{2}\right)^2$

Exponents: $x^n x^m = x^{n+m}$, $\frac{x^n}{x^m} = x^{n-m}$, $(x^n)^m = x^{nm}$

Logarithms: If $y = \log_b a$, then $b^y = a$

- common log: $\log_{10} \triangleq \log$
- log base 2: $\log_2 \triangleq \lg$
- natural log: $\log_e \triangleq \ln$
- LOG RULES:
  - $\log_b 1 = 0$ $(b>0)$
  - $\log_a a^x = x$ $\quad$ • $a^{\log_a x} = x$
  - $\log_b(xy) = \log_b x + \log_b y$
  - $\log_b\left(\frac{x}{y}\right) = \log_b x - \log_b y$
  - $\log_b x^n = n \log_b x$
  - $\log_a x = \frac{\log_b x}{\log_b a}$

Combinatorics

- a k-combination is the number of unordered subsets of k out of n elements (no rep) $\binom{n}{k} = {}_nC_k = \frac{n!}{(n-k)!\,k!}$
- a k-permutation is the number of ordered subsets of k out of n elements w/ no repetition ${}_nP_k = \frac{n!}{(n-k)!}$

Asymptotics

$f = \Theta(g)$	f grows at the same rate as g	$(=)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0, \infty$
$f = O(g)$	f grows no faster than g	$(\le)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$
$f = \Omega(g)$	f grows at least as fast as g	$(\ge)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$
$f = \omega(g)$	f grows faster than g	$(>)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$
$f = o(g)$	f grows slower than g	$(<)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Θ: $f \in \Theta(g)$ iff $f \in O(g)$ and $g \in O(f)$

O: $f \in O(g)$ iff $\exists c>0$ and $\exists n_0>0$ s.t. $|f(n)| \le c|g(n)|$ $\forall n > n_0$

Ω: $f \in \Omega(g)$ iff $\exists c>0$ and $\exists n_0$ s.t. $|f(n)| \ge c|g(n)|$ $\forall n > n_0$ $(g \in O(f))$

Stirling's Approximation: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha}$ $\ln n! \approx n\ln n - n$

Recurrences

1) Substitution Method: guess a solution & prove w/ induction

e.g. Solve $T_1(n) = T_1(n-1) + \overset{\text{const}}{\Theta(1)}$ with substitution method

Guess: $T_1(n)$ is $\Theta(n)$: $T_1(n) \le cn$ for some constant c, $n \ge n_0$

PROVE $T_1(n) \in O(n)$
- Base Case: $n=1$ $\quad T_1(1) = \text{const} + 1 \le c * 1$ ✓
- Inductive hypothesis: assume $T_1(n) \le cn$ for some c

PROVE $T_1(n+1) \le c(n+1)$

$T_1(n+1) = T_1(n) + 1$ ← for recurrence

$\le cn + 1$

$\le c \cdot n + c$

$\le c(n+1)$

PROVE $T_1(n) \in \Omega(n)$ ✓
- Base Case: $n=1$, $T_1(1) = \text{const}+1 \ge c*1$
- Induct. Hyp: assume $T_1(n) \ge cn$ for some c

PROVE $T_1(n+1) \ge c(n+1)$

$T_1(n+1) = T_1(n) + 1$

$\ge cn + 1$

$\ge cn + c$

$\ge c(n+1)$

e.g. Prove by induction that $T(n) \le T(n/5) + T\left(\frac{7n}{10}\right) + \Theta(n)$ is $\Theta(n)$

i.e. $T(n) \le cn$ for some c

Assume that $T(n) \le ck$ for all $k < n$

Prove for $k = n$

$T(n) \le c\left(\frac{n}{5}\right) + c\left(\frac{7n}{10}\right) + an = c\left(\frac{9n}{10}\right) + an$

$= n\left(\frac{9c}{10} + a\right) \le c \cdot n$ ✓ IF C large enough

2) Iteration Method: convert a sequence to a sum by iterating

e.g. Solve $T_2(n) = 2T_2\left(\frac{n}{2}\right) + 1 = 2\left[2T_2\left(\frac{n}{4}\right) + 1\right] + 1$ w/ iteration

$= 4T_2\left(\frac{n}{4}\right) + 2 + 1 = 4\left[2T_2\left(\frac{n}{8}\right) + 1\right] + 2 + 1 = 8T_2\left(\frac{n}{8}\right) + 4 + 2 + 1 =$

$= nT_2(1) + \dots + 8 + 4 + 2 + 1$

$= \sum_{i=1}^{\log_2 n} 2^i = \frac{2^{\log_2 n + 1}}{2-1} - 1 = 2n - 1$, so $T(n) \in \Theta(n)$

e.g. Solve w/ iteration: $T(n) = T(n-1) + 3$

$= [T(n-2) + 3] + 3$

$= [[T(n-3) + 3) + 3] + 3$

$T(n) = T(0) + \sum_{i=1}^{n} 3 = T(0) + 3n = \Theta(3n)$

$\to T(n) \in \Theta(n)$

e.g. Solve with iteration: $T(n) = 8T(n/4) + n^2$, $T(1) = 1$

$= 8\left[8T(n/4) + (n/2)^2\right] + n^2 = 8^2 T(n/4) + 8(n/2)^2 + 8(n)^2 = 8\left[8\left[8T(n/8) + \left(\frac{n}{4}\right)^2\right] + \left(\frac{n}{2}\right)^2 + n^2\right.$

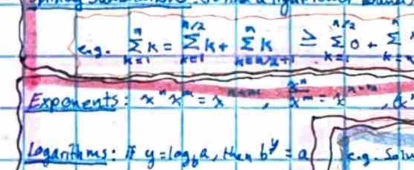
$= 8^3 T\left(\frac{n}{8}\right) + 8^2\left(\frac{n}{4}\right)^2 + 8(n/2)^2 + 8^0 n^2 = 8^{\log_2 n} T(1) + 8^{\log_2 n - 1}(n/2n)^2 + 8^{\log_2 n - 2} + \dots + 8^2(n/4)^2 + 8(n/2)^2 + n^2$

$= \left(\sum^{\log_2 n - 1} 8^i (n/2^i)^2\right) + 8^{\log_2 n} = n^2(n-1) + n^3 = \Theta(n^3)$

3) Recursion Tree Method: write the recurrence tree, sum the rows

e.g. Find runtime using the recursion tree method:

$T(n) = 2T(n/2) + n$



$\log_2 n$
T(n)
T(n/2) T(n/2)
T(n/4) T(n/4) T(n/4) T(n/4)
T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8)
↳ stop when reach base case (T(1))
T(1)...
$2^{\log_2 n} = n$ nodes

Non recurrent components:

n

$\frac{n}{2} + \frac{n}{2} = n$

$\frac{n}{4} + \dots = 4\left(\frac{n}{4}\right) = n$

$\vdots$

n

COST: $n \cdot T(1)$ $\qquad n \cdot \log_2 n$

Total Cost: $T(n) = n \cdot T(1) + n\log_2 n = cn + n\log_2 n \to \therefore T(n) \in \Theta(n\log n)$

4) Master Method: use for recurrences of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$; $a \ge 1$, $b > 1$

a) $f(n)$ is $O(n^{\log_b a - \epsilon})$ for $\epsilon > 0$ $\Rightarrow T(n) \in \Theta(n^{\log_b a})$

b) $f(n)$ is $\Theta(n^{\log_b a})$, $T(n) \in \Theta(n^{\log_b a} \log n)$

c) $f(n)$ is $\Omega(n^{\log_b a + \epsilon})$, $af\left(\frac{n}{b}\right) \le c \cdot f(n)$ for some $c<1$, $n > n_0$ $\Rightarrow T(n)$ is $\Theta(f(n))$

e.g. $T(n) = 2T(n/2) + 1$ (MASTER METHOD)

Compare $n^{\log_b a} = n^{\log_2 2} = n^1 = n$, $f(n) = 1$

CASE 1: $T(n)$ is $\Theta(n^{\log_b a})$, so $\Theta(n)$

e.g. $T(n) = T(n/2) + 1$ (MASTER METHOD

Compare $n^{\log_b a} = n^{\log_2 1} = 1$ and $f(n) = 1$

1 is $\Theta(1) \Rightarrow$ case 2

$\Rightarrow T(n)$ is $\Theta(n^{\log_b a} \log n)$, so $\Theta(\log n)$

e.g. $T(n) = 9T\left(\frac{n}{3}\right) + n^3 \to$ MASTER

Compare $n^{\log_b a} = n^{\log_3 9} = n^2$

and $f(n) = n^3$

Case 3? n^3 is $\Omega(n^{2+\epsilon})$ ✓

$9\left(\frac{n}{3}\right)^3 \le c(n^3)$, $c = 3$, $n_0 = 9$

So, $T(n)$ is $\Theta(n^3)$

e.g. $T(n) = 2T(n/2) + n\log n$

Compare: $n^{\log_b a} = n^{\log_2 2} = n$

to $f(n) = n\log n$

$f(n) = n\log n$ is $\Omega(n)$ — case 3?

Is $n\log n$ $\Omega(n^{1+\epsilon})$? NO.

CAN'T USE MASTER METHOD

5) DOMAIN/ RANGE TRANSFORMATION

Converts division/mult of n by a factor to subtraction/division

e.g. Domain/Range Transformation

$T(n) = T\left(\frac{n}{2}\right) + 1$ (Case 2)

Transform domain: $n = 2^i$

$T(2^i) = T\left(\frac{2^i}{2}\right) + 1 = T(2^{i-1}) + 1$

Define new function: $S(i) = T(2^i)$

So, $S(i) = S(i-1) + 1$

SOLVE W/ ITERATION

$S(i) = (S(i-2) + 1) + 1) + 1$

$= S(i-2) + 2 =$

$(S(i-3) + 1) + 2$

$= S(1) + i \to \Theta(i)$

Backtracking $S(i) = T(2^i) = c \cdot i$

$T(n) = c\log_2 n$

$T(n)$ is $\Theta(\log n)$

$T(n) = 4T\left(\frac{n}{2}\right) + n^7$, ITERATION

$= [T(n-2) + (n-1)^7] = [T(n-3) + (n-2)^7] + (n-1)^7 + n^7 = \sum_{k=0}^{n} k^7$

BOUNDING BY INTEGRALS: $\underbrace{\int_{x=0}^{n} x^7\,dx}_{x^8/8} \le \sum_{i=1}^{n} i \le \underbrace{\int_1^{n+1} x^7\,dx}_{(n+1)^8} \Rightarrow T(n)$ is $\Theta(n^8)$

Searching

- **Find min:**
  - sort: $\Theta(n\log n)$
  - go over array once, compare to current min – $\Theta(n)$
- **Find min and max:**
  - store cur min, cur max, loop through – $\Theta(n)$
  - OR compare two elems in array, find which is bigger, compare that to max, other to min, $3\lceil n/2 \rceil$ comparisons instead of $2n$
- **SELECTION (kth smallest element), Select(k,n)**
  1. Divide elements into $\lceil n/5 \rceil$ groups of 5 elements $\Theta(1)$
  2. Find median of each group $\Theta(n)$
  3. Recursively find median of medians, M $T(n/5)$
  4. Partition A(pivot) on M $\Theta(n)$
  5. Recur on the appropriate partition:

```
Let M be the jth element
if k=j, return M
else if k<j, call select recursively to find kth
    element on left partition
else call select recursively to find the (k-j)th
    element on right partition
```

OVERALL ~~[illegible]~~ Linear RUNTIME

e.g. If we have: 314122485421 (5) 6 9 6 8 9 7 8 6 → j=14
- select the THIRD ELEMENT
  - 3<14, so recurse on left partition, 3rd elem on left
- select the 15th ELEMENT
  - 15>14, so recurse on right partition 15-14=1st element

Binary Search Trees

- **Tree:** one root node, nodes have child nodes, all nodes except root have one parent
- ACYCLIC and FULLY CONNECTED

- **Binary tree:** every node has 0,1, or 2 children
- **Complete binary tree:** all levels have max number of nodes (except bottom level → filled left to right)
- **Full Binary tree:** each node is a leaf or has two children

- **FULL BINARY TREE THEOREM:** the number of leaves in a non-empty, full binary tree is equal to the number of internal nodes in the tree +1
- any tree with n nodes has n-1 edges
- **SEARCH(x, val):** x is a pointer to the current node, initially root

```
if(x==null)
    return // give up
else if key(x)> value
    Search(left(x), value)
else
    search(right(x), value)
```

Runtime: $\Theta(\text{height})$
- worst case $\Theta(n)$
- best case $\Theta(\log n)$

Find min: go left until you run out of nodes
Find max: go right until you run out of nodes
Insert: Search to find where it should be inserted, insert where we fell off the tree $\Theta(\text{height})$
Successor: next largest element after v in the tree $\Theta(\text{height})$

```
if there is a right subtree
    return its minimum
else
    go up until you get a node w/ a right parent
    return parent
```

Delete:

```
if v is a leaf, remove v
if v has one child, connect v's parent to v's child
else // v has 2 children
    Swap v w/ successor (v)
    delete v in its new location
```

$\Theta(n)$ worst case
$\Theta(h)$ typically

Hashing

m = table size, k = key   frac( ) = fractional portion

- **division hash:** m a large prime, hash val is k%m
  - Uniformly random numbers will map uniformly to table
- **multiplication hash:** $h(k) = \text{trunc}(m * \text{frac}(k*A))$, $0<A<1$
  - slower than division hash, BOTH SIMPLE UNIFORM

COLLISION RESOLUTION:
- **Open hashing:** each item in table a linked list
  - **insert:** insert to head, $\Theta(1)$ → if perf. dist. $\Theta(n/m)$
  - **search/delete:** worst case all elements in one place, $\Theta(n)$
- **Load factor:** of a hash, $\alpha = n/m$, average length of each chain
- **THEOREM:** SUH tables w/ chain-based collision resolution takes $\Theta(1+\alpha)$ to search, * try to use $\alpha \approx 1$ *
- **closed hashing:** (open addressed hash)
  - each slot holds one element, α can't exceed 1, ideally $\alpha < 0.5$
  - **probing sequence:** if hash position is free, place item else, go to next pos. in sequence
  - **Linear Probing:** $h_i(k) = [h'(k)+i] \bmod m$, $i=0 \ldots m-1$ ($h'(k)$ given)
    - PROBLEM: primary clustering, interferes w/ neighboring hashes, results in long sequence of probes
  - **Quadratic Probing:** $h_i(k) = [h'(k) + c_1 i + c_2 i^2] \bmod m$
    - PROBLEM: secondary clustering, interference b/w items w/ same initial hash, can't always get unused slots
  - **Double hashing:** use secondary hash for probing sequence
    - $H_i(k) = (h_1(k) + i h_2(k)) \bmod m$
    - solves secondary clustering, larger comp. overhead
  - **Search:** apply hash, search through probe sequence until you find an empty slot
  - **delete:** LAZY DELETE: don't remove, just mark deleted
- **Rehashing:** periodically creating a larger hash table and reinserting elements

BLOOM FILTER: provides a compact way of checking membership to a set
- maps a set S into an array of bits B (bitmap) of size m → for $i=1..k$
- **insert x into S:** compute $h_1(x), h_2(x), \ldots h_k(x)$, set $B[h_i(x)]=1$
- **checking x's membership in S:** compute $h_1(x) \ldots h_k(x)$
  - if not all corresponding $B[h_i(x)]$'s have 1, x is NOT a member of S
  - if all $B[h_i(x)]$'s are 1, x MAY be a member of S
- CANNOT DELETE ELEMENTS
- ★ cell sum: xor of id of key, inc count (XOR of all ID's in the cell)

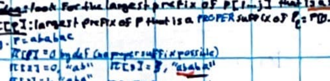
STRING MATCHING

- **Problem:** finding a pattern, $P = P_0 P_1 \ldots P_{m-1}$ in a text, $T = T_0 T_1 \ldots T_{n-1}$ ($m \ll n$)
- **Naive algorithm:**

```
For s=0, to n-m:
    if P[0....m-1] == T[s ... s+m-1]
        output "found it!"
```

Runtime $\Theta(n \cdot m)$

- **DETERMINISTIC FINITE AUTOMATION (FSM):**
  - Σ - input alphabet, S - set of states, $S_0 \in S$ - initial state, $F \subseteq S$ - set of finite states
  - $\delta: S \times \Sigma \to S$, transition function
- **Runtime:** Processing (building) $\Theta(m \cdot |\Sigma|)$, computing (searching) $\Theta(n)$
- e.g. FSM for "ababa" → $\Sigma = \{a,b\}$, $S_0 = 0$, $S = \{0,1,2,3,4,5,6\}$, $F = \{6\}$



KMP: an automaton-less version of pattern recognition
- **Idea:** look for the largest prefix of P[1...j] that is a suffix of P[1...j]
- $\pi[q]$: largest prefix of P that is a PROPER suffix of $P_q = P[1..q]$

e.g. P = ababac
- $\pi[1] = 0$ by def (no proper suffix possible)
- $\pi[2] = 0$, "ab"
- $\pi[3] = 1$, "aba"
- $\pi[4] = 2$, "abab"
- $\pi[5] = 3$, "ababa"
- $\pi[6] = 0$ "ababac"
- $\pi = [0\,0\,1\,2\,3\,0]$

CODE:

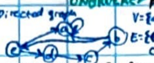
```
1) ComputePrefix(P) // compute π
    π[1]=0, k=0
    for q=2 to m (m=len(p))
        while k>0, P[k+1] != P[q]
            k = π[k]
        if P[k+1] = P[q]
            k++
        π[q]=k

2) KMP(P, T)        |P|=m, |T|=n
    computePrefix(P)
    q=0
    for i=1 to n
        while q>0 and P[q+1] != T[i]
            q=π[q]
        if P[q+1] = T[i]
            q++
        if q == m
            PATTERN MATCHES
            q=π[q]
```

Runtime:
- Preprocessing: $\Theta(m)$
- Pattern matching: $\Theta(n)$, so $\Theta(n+m)$

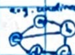
Graphs

- **directed graph:** G(V,E) such that V is a set of vertices and $E \subseteq V \times V$ is a set of edges, each edge is an ORDERED set of vertices (v,w) where $v,w \in V$
- **undirected graph:** G=(V,E) such that V is a set of vertices, E a set of edges, each edge an UNORDERED pair of vertices

e.g. Directed graph — V={a,b,c,d,e}, E={(a,b),(a,a),(c,a),(b,c),(c,d),(d,e),(e,b)}



e.g. undirected graph — V={a,b,c,d,e}, E={{a,b},{b,c},{c,d},{d,e},{a,c},{e,a}}

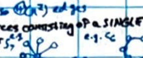


- **degree:** the number of edges directly connected to a node
  - **in-degree** and **outdegree**: the number of edges pointed into or out of a node
- **adjacent vertices:** vertices connected by an edge
- **weighted graph:** directed or undirected graph w/ weight labels on edges

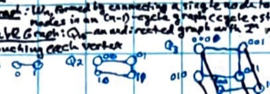
TYPES OF GRAPHS
- **complete graph:** on n vertices – K_n: an undirected graph w/ n vertices & an edge b/w any two vertices

e.g. K_1, K_2, K_3, K_4 … K_n has $\frac{n(n-1)}{2}$ edges, so $\Theta(n^2)$ edges

- **cycle graph:** on n vertices – C_n: undirected graph w/ n vertices consisting of a SINGLE cycle
- **Star graph:** a tree (S_n) with one internal node and n leaves
- **Wheel Graph:** W_n, formed by connecting a single node to all nodes in an (n-1)-cycle graph (cycle + star)

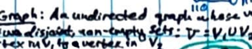


- **Hypercube Graph:** Q_n – an undirected graph with 2^n vertices, 2^{n-1} edges, and n edges touching each vertex



Connect vertices w/ Hamming distance of one → # of bit flips

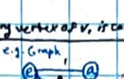
- **Bipartite Graph:** An undirected graph whose vertex set V can be partitioned into two disjoint, non-empty sets: $V = V_1 \cup V_2$ such that every edge connects a vertex in V_1 to a vertex in V_2



- **Complete Bipartite Graph:** $K_{n,m}$: $|V_1| = n$, $|V_2| = m$, every vertex of V_1 is connected to every vertex of V_2 e.g. $K_{4,2}$

GRAPH REPRESENTATIONS

e.g. Graph



1) **Edge list:** array/list/vector of edges and overall number of vertices
   - e.g. [{a,b}, {b,c}, {d,c}, {d,e}, {e,a}, {b,e}]
   - *use ordered pairs for directed graphs*
   - **Space:** to store V vertices and E edges, $\Theta(E)$
   - **adjacency time:** given the name/number of two vertices, time to determine adjacency: $\Theta(E)$
   - **add edge:** $\Theta(1)$
2) **Adjacency list:** Table of vertex & its neighbors in a sorted list (array of sorted/unsorted linked lists/arrays)

vertex	neighbors
a	be
b	ace
c	bd
d	ce
e	abd

   - **Space:** $\Theta(V+E)$ ie, neighbors list
   - **Adj time:** sorted: $\Theta(\log V)$; unsorted: $\Theta(V)$
   - **Add edge:** sorted: $\Theta(V)$; unsorted: $\Theta(1)$
   - *Directed graphs: list neighbors w/ outgoing edges from each vertex*
3) **Adjacency Matrix:** Rows & columns are vertex names for source & destination – contains 1 for an edge, 0 for no edge
   - For directed graph, '1' only for directed edge
   - For weighted graph, replace 1 with weight, 0 w/ NULL

	a	b	c	d	e
a	0	1	0	0	1
b	1	0	1	0	1
c	0	1	0	1	0
d	0	0	1	0	1
e	1	1	0	1	0

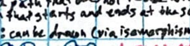
   - space: $\Theta(V^2)$
   - adjacency time: $\Theta(1)$
   - add edge: $\Theta(1)$
4) **Incidence Matrix:** Rows are vertex numbers, columns are edge names/numbers, contains '1' for each edge from this vertex

	1	2	3	4	5	6
a	1	1	0	0	0	0
b	0	1	1	0	1	0
c	0	0	0	0	1	1
d	0	0	0	1	0	1
e	1	0	1	1	0	0

   - space: $\Theta(V \cdot E)$
   - adjacency time: $\Theta(E)$
   - add edge: $\Theta(V)$
   - Directed graph: incoming 1, outgoing -1

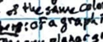
GRAPH RELATIONSHIPS

- **isomorphism:** graph $G_1 = (V_1, E_1)$ is (can) isomorphic to graph $G_2 = (V_2, E_2)$ if there is a 1 to 1 function $f: V_1 \to V_2$ such that $(a,b) \in E_1 \leftrightarrow (f(a), f(b)) \in E_2$

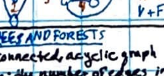


- **path:** a sequence of edges joined end-to-end; if the graph is directed, edges must point in the same direction
  - e.g. one path: ({3,1}, {1,2}, {2,5}) = {3125}

- **simple path:** a path that does not reuse an edge, e.g. {3125} simple, {125243} is not
- **cycle:** a path that starts and ends at the same vertex, e.g. {12431}
- **planar graph:** can be drawn (via isomorphism) on a plane w/o crossing edges
  - e.g. ✓ PLANAR



- **Kuratowski's Theorem:** a graph is planar iff it does NOT contain a subdivision of K_5 or $K_{3,3}$
- **Graph coloring:** an assignment of colors to vertices such that no edge connects two nodes of the same color
- **chromatic number:** of a graph is the fewest number of colors required to color it
- **4 COLOR THEOREM:** any planar graph can be colored with 4 colors
  - ★ A two-color graph must be bipartite by definition
  - minimum # of colors ≥1, ≤V
  - greedy algorithms use local optimization – suboptimal solution
  - Brute force: try all possible combinations, EXPONENTIAL RUNTIME
- **Eulerian path:** a path that passes through every edge exactly once
- **Eulerian cycle:** an Eulerian path that starts & ends at the same vertex
- **Handshaking Theorem:** if G=(V,E) is an undirected graph, then $\sum_{v \in V} \deg(v) = 2|E|$
  - **corollary:** always have an even # of vertices w/ odd degree
- a graph is **connected** iff there is a path b/w any two vertices in the graph
- **Euler's Graph Theorem:** a graph has an Eulerian cycle iff it is connected and every vertex has an even degree; a graph has an Eulerian path iff it is connected and at most two vertices have an odd degree
- a path is **Hamiltonian** if it touches every vertex exactly once
- a **Hamiltonian cycle** is a Hamiltonian path w/ same start & end point
- **Hamiltonian cycle theorem:** a graph has a Hamiltonian cycle has at least $n \geq 3$ vertices and every vertex has a degree $\geq n/2$
- **Euler's Formula:** If a graph is planar, $V + F = E + C + 1$ (# vertices, # faces, # of edges, # connected components)



10 + 5 = 12 + 2 + 1 — V + F = E + C + 1

TREES AND FORESTS

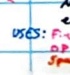
- **Tree:** a connected, acyclic graph
- **Theorem:** the number of edges in a tree is V-1 (via Euler's)
- **Forest:** an acyclic graph (E=V-C)
- **Theorem:** a graph is a tree iff there is exactly one path (not EDGE) b/w any two vertices
- **Tree traversal:**
  - **pre-order:** visit root, left sub, right sub, recurse
  - **post-order:** left subtree, right subtree, root
  - **in-order:** left subtree, root, right subtree, recurse

GRAPH ALGORITHMS: TRAVERSAL (to gain info)

BFS: Breadth-First Search
- **idea:** start with a node and visits its neighbors, their neighbors, etc.
- **data structure:** graph & a queue
  - 3 colors, white = unvisited, gray = working on, black = done with
- **Runtime:** $\Theta(V+E)$ by aggregate analysis $\Theta(V+E)$
- if you run an unconnected graph → only reach connected portions

```
BFS(G,S)
    color all vertices white, S gray
    d(v)=∞, d(s)=0, π(v)=null
    enqueue(S)
    while queue is not empty
        h = head of queue
        for each neighbor n of H
            if n is white
                color gray
                enqueue n
                d(n) = d(h)+1
                π(n)=h
        dequeue(h)
        color h black
```

Queue: ~~A~~BCD — DONE W/ FIRST LOOP
Queue: ~~B~~CDE — DONE W/ SECOND LOOP



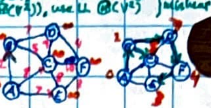
Queue: ~~C~~DEF — DONE — Now just empty queue

USES: Find shortest path in terms of # of edges: d[V]; Spanning tree (π[v] pred. mat.)

GRAPH ALGORITHMS: Minimum Spanning Tree

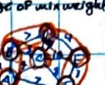
- **Minimum spanning tree:** spanning tree w/ minimum sum of weights for edges
- **Bridge Property:** Given two disjoint sets of vertices, $V_1 \cap V_2 = \emptyset$, the lightest edge b/w V_1 and V_2 must be part of some MST
- **Prim's Algorithm:** greedy algorithm for finding an MST in a connected, weighted, undirected graph
  - **Idea:** repeatedly adding a light edge from A to V-A to the MST
  - **Data structures:** priority queue: w(u,v), weight of edge {u,v}; π[v] pred. array; key[v]
  - **Runtime:** Sparse Graph (E is $\Theta(V)$) use heap $\Theta(V\log V)$; Dense Graph (E is $\Theta(V^2)$), use LL $\Theta(V^2)$ — LL: $\Theta(V^2)$, minheap: $\Theta(E\log V)$

```
Prim(G, root)
    key[v] = ∞ for all v, 0 for root
    Q ← all vertices
    while Q≠∅
        u ← extract min (Q)
        for each v adj to u
            if v∈Q and w(u,v) < key[v]
                key[v] ← w(u,v)
                π[v] ← u
```

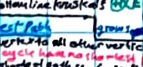


- **Kruskal's Algorithm:** idea is to repeatedly add an edge of min weight that DOES NOT form a cycle to the MST

```
1) Make-set(v) for each v∈V
2) Sort edges in non-decreasing order
3) For each edge(u,v) in non-decreasing weight order
    if find-set(v) ≠ find-set(u)
        union (u,v)
RESULTING SET IS MST!
```



- **Disjoint Set Data Structure:**
  - OPERATIONS:
    - Makeset(x) – make a component containing x only
    - find-set(x) – returns representative of x
    - union(x,y) – joins components of x and y
  - **Implementation: Linked lists**
    - represent each set as a linked list, 1st element is the rep and all members point to it
    - RUNTIMES: makeset $\Theta(1)$, find-set $\Theta(1)$, union $\Theta(n)$
    - OVERALL KRUSKAL: $\Theta(VE + E\log E)$
  - **Implementation: Linked lists w/ Weighted Heuristic**
    - always redo pointers of SHORTER list to LONGER list
    - Runtimes: union now $\Theta(\log n)$, overall Kruskal $\Theta(E\log V)$
  - **Implementation: Disjoint Forest**



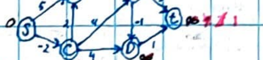
    - if set[i]=i, i is root
    - if set[i]=j, j is i's parent
    - path compression: during find-set operation, point every node I pass through to the root
    - Runtime: weird for FS/UN, but bottom line kruskal's $\Theta(E\alpha(V))$ → inverse Ackerman, grows slower than log

GRAPH ALGORITHMS: Single Source Shortest Path

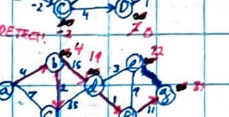
- **goal:** find shortest path from a single vertex to all other vertices
- ★ caveat: graphs w/ negative weight cycle have no shortest path
- **optimal substructure property:** for any shortest path u…i…v, the subpath u…i is the shortest path as well
- **Representation:** π[v]: v's predecessor in the shortest path from S; d[v]: upper bound on shortest path from S to v

```
Relax: (u,v,w(u,v))
    if d[v] > d[u] + w(u,v)
        d[v] = d[u] + w(u,v)
        π[v] = u
```

e.g. Relax(u,z,10) — d[z] = 75 = 60+10 = 70 = d[z]; π[z] = u

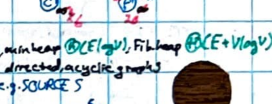


```
Bellman-Ford(G,S):
    d[v]=∞; d[s]=0; π[v]=null
    repeat V-1 times:
        relax each edge
    relax each edge again
        if any relaxation is constructive,
        output NEG. WEIGHT CYCLE    CAN DETECT!
```



- **Runtime:** VE const relaxations, so $\Theta(VE)$

```
Dijkstra(G,S):
    d[v]=∞, d[s]=0, π[v]=null
    add all v to Q, indexed by d
    while Q≠∅
        u ← extract min (Q)
        for each v in Q adjacent to u
            relax(u,v, w(u,v))
```

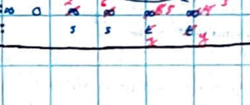


- **Runtime:** unordered LL $\Theta(E+V^2)$, minheap $\Theta(E\log V)$, Fib heap $\Theta(E+V\log V)$
- **DAG shortest Path:** only works for directed, acyclic graphs

```
DAG(G,S):
    d[v]=∞, d[s]=0, π[v]=null
    for each v in TOPOLOGICAL ORDER
        for each u adjacent to v
            relax(u,v, w(u,v))
```

- **Runtime:** $\Theta(V+E)$

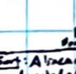
e.g. SOURCE S



Sadie Allen
U02945217

DFS: Depth-First Search
- **idea:** for each node, go as far as possible in one direction, backtrack & restart
- **data structures:** STACK (recursion), same node coloring
- **Runtime:** $\Theta(V+E)$, aggregate analysis

```
DFS(G)
    color all vertices white
    π(v)=null, time=0
    for each white vertex V
        DFS_visit(G,V)

DFS_visit(G,V)
    color v gray
    for each node n adjacent to v
        if n is white
            π(n)=v
            DFS_visit(G,n)
    color v black
    f(v) = time++;
```

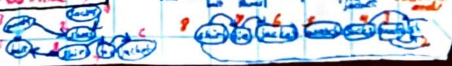


```
DFS_Visit(G,A)
  DFS_Visit(G,B)
    DFS_Visit(G,C)
      DFS_Visit(G,D)
      Done
      DFS_Visit(G,E)
      Done
    Done
  Done
Done
```

Stack

★ works for unconnected graphs

Topological Sort: A linear ordering of vertices in a graph s.t. for every directed edge from u to v, u comes before v



~~[illegible]~~ Graph Algorithms: All Pairs Shortest Path

Goal: Find shortest path b/w any two vertices

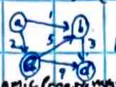
- **ITERATED SINGLE-SOURCE SOLUTIONS:**
  - run single source shortest path for all vertices in graph!
  - Dijkstra's: LL - $\Theta(E+V^2) \rightarrow \Theta(EV+V^3)$
    - (not [illegible] for neg weight [illegible]) min heap - $\Theta(E\log V) \rightarrow \Theta(VE\log V)$
    - Fib heap - $\Theta(V\log V+E) \rightarrow \Theta(V^2\log V+VE)$
  - Bellman-Ford: $\Theta(VE) * V \rightarrow \Theta(V^2E)$
    - On a dense graph: $\Theta(V^4)$
  - Representation:
    - Input: graph
    - Output:
- distance matrix D (distance b/w pairs, generalization of d[V])

$$d_{i,j} = \begin{cases} 0, & i=j \\ \infty, & j \text{ not reachable from } i \\ \text{weight of shortest path from } i \text{ to } j, & \text{all other cases} \end{cases}$$

- Predecessor Matrix Π (second to last path b/w pairs, generalization of π)

$$\Pi_{ij} = \begin{cases} \text{NIL}, \emptyset, & i=j, j \text{ not accessible from } i \\ \text{pred in shortest path from } i \text{ to } j & \text{otherwise} \end{cases}$$

e.g.



$$D = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 2 & 4 \\ b & \infty & 0 & \infty & 3 \\ c & \infty & 5 & 0 & 7 \\ d & \infty & \infty & \infty & 0 \end{array} \qquad \pi = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \emptyset & a & a & b \\ b & \emptyset & \emptyset & \emptyset & b \\ c & \emptyset & c & \emptyset & b \\ d & \emptyset & \emptyset & \emptyset & \emptyset \end{array}$$

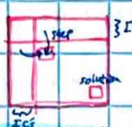
Dynamic Programming: Solve a complex problem by breaking it down into sub problems as follows:
1) Determine initial conditions
2) Define rules/step to solve sub problem
3) Reach a solution!

e.g. 1D Dynamic programming: Fibonacci
- IC: Fib(0) = Fib(1) = 1
- Rule: Fib(i) = Fib(i-1) + Fib(i-2)
- Solution: Fib(n) for some n > 1

solution
IC's

Runtime: $\Theta(n)$

2D Dynamic Programming



★ Need sufficient initial conditions ★ to reach solutions!

Floyd-Warshall Algorithm

- D^0 = adjacency matrix, D^V = solution
- number all vertices 1, 2, ... V
- $d_{i,j}^m$ = the weight of the shortest path from i to j that only uses v/ numbers 1...m
- This path either:
  1) Does not include vertex m
  2) Does include m and is the sum of the paths from i to m and from m to j

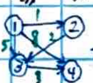
Step: $d_{i,j}^m = \min\{\underbrace{d_{i,j}^{m-1}}_{\Theta(1)}, \underbrace{d_{i,m}^{m-1} + d_{m,j}^{m-1}}_{\Theta(1)}\}$ → $\Theta(1)$

Overall runtime: $\Theta(1)$ per cell, V^2 ← # of cells, $\Theta(V)$ ← # of steps → $\boxed{\Theta(V^3)}$

π? $\Pi_{i,j}^0 = \begin{cases}\emptyset & \text{if } i=j, i \text{ not adjacent to } j \\ i & \text{else}\end{cases}$

$\Pi_{i,j}^m = \Pi_{i,j}^{m-1}$ if (1) or $\Pi_{m,j}^{m-1}$ if (2)

e.g.



$$\Pi^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & \emptyset & 1 & 1 & 1 \\ 2 & \emptyset & \emptyset & 2 & \emptyset \\ 3 & \emptyset & \emptyset & \emptyset & 3 \\ 4 & \emptyset & \emptyset & \emptyset & \emptyset \end{array} \ldots \Pi^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & \emptyset & 1 & 2 & 3 \\ 2 & \emptyset & \emptyset & 2 & 3 \\ 3 & \emptyset & \emptyset & \emptyset & 3 \\ 4 & \emptyset & \emptyset & \emptyset & \emptyset \end{array}$$

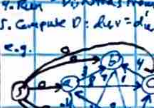
$$D^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 5 & 9 \\ 2 & \infty & 0 & 2 & \infty \\ 3 & \infty & \infty & 0 & 3 \\ 4 & \infty & \infty & \infty & 0 \end{array} = D^1 \quad D^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 3 & 9 \\ 2 & \infty & 0 & 2 & \infty \\ 3 & \infty & \infty & 0 & 3 \\ 4 & \infty & \infty & \infty & 0 \end{array} \Rightarrow D^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 3 & 6 \\ 2 & \infty & 0 & 2 & 5 \\ 3 & \infty & \infty & 0 & 3 \\ 4 & \infty & \infty & \infty & 0 \end{array} \Rightarrow D^4 = D^3$$

Johnson's Algorithm

- Single-source shortest path on V (do it V times)
  1) Dijkstra's w/ Fib heap: $\Theta(V\log V + E) * V$ ← worst case V^3 for dense
  2) Bellman-Ford: $\Theta(EV) * V$ ← worst case V^4, DETECTS neg cycles

Goal of Johnson's: combine robustness of BF w/ performance of Dijkstra
1. Add a new source S with weight 0 edges to all other vertices
2. Run BF from S to all other vertices to get d[V], report neg weight cycle
3. Reweigh each edge: $\omega'(u,v) \leftarrow \omega(u,v) - d[v] + d[u]$
4. Run Dijkstra's from each vertex to get D' and π
5. Compute D: $d_{u,v} = d'_{u,v} + d[v] - d[u]$ $\Theta(V^2\log V + VE)$

e.g.



$$D' = \begin{array}{c|ccccc} a & 0 & 2 & 2 & 2 & 0 \\ b & 2 & 0 & 0 & 0 & 2 \\ c & 2 & 0 & 0 & 0 & 2 \\ d & 2 & 0 & 0 & 0 & 2 \\ e & 4 & 2 & 2 & 2 & 0 \end{array} \qquad D = \begin{array}{c|ccccc} a & 0 & 1 & -3 & 2 & -4 \\ b & 3 & 0 & -4 & 1 & -1 \\ c & 7 & 4 & 0 & 5 & 3 \\ d & 2 & -1 & -5 & 0 & -2 \\ e & 3 & 5 & 1 & 6 & 0 \end{array}$$

$$\pi = \begin{array}{c|ccccc} a & \emptyset & c & d & e & a \\ b & d & \emptyset & d & b & a \\ c & d & c & \emptyset & \emptyset & a \\ d & d & c & d & \emptyset & a \\ e & d & c & d & e & \emptyset \end{array}$$

Template Class

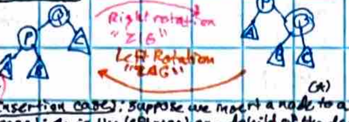
numCell.h
```
#include <string>
template <class T>
class numCell{
   private:
      T storedValue;
   public:
      numCell();
      numCell(T);
      T read();
      void write(T);
      numCell &swap(numCell &);
      numCell & operator=(numCell);
      numCell operator+(numCell &);
};
template <class T>
numCell<T>::numCell()
{ //nothing
}
template<class T>
numCell<T>::numCell(T val)
{
   this->storedValue=val;
}
```
ONLY CHANGES: replace all "numCell"'s w/ numCell<T>, all ints w/ Ts

main.cpp
```
int main()
{ numCell<int> num1(46);
  numCell<string> str1("hi");
  //use
  return 0;
}
```

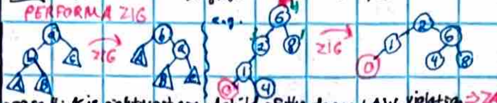
Balanced Search Trees

- binary search trees: binary tree; for any node x, all nodes in left subtree of x are ≤ x, all nodes in right subtree of x are ≥ x
- Runtime for insertion in BST w/ n nodes: $\Theta(h)$, $\Theta(n)$ worst case, $\Theta(\log n)$ best case
- AVL Tree: a balanced BST where for every internal node v, the height of the left & right subtrees differ by 1 at MOST. height = distance from farthest leaves
  - Maximum height: $AVL_h = AVL_{h-1} + AVL_{h-2} + 1 \geq Fib(h) \approx \emptyset^h$
    $AVL_h \geq \emptyset^h \Rightarrow \log_{\emptyset} AVL_h \geq h$
    AVL has logarithmic height w/ respect to # of nodes
    Log(n)
- Insert: same as BST, fix AVL violations (still $\Theta(\log n)$ since rotations are all constant time

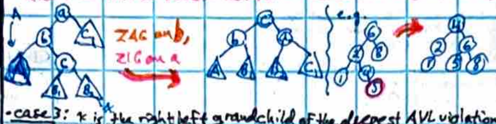
ROTATIONS



Insertion cases: Suppose we insert a node to a valid AVL tree
- case 1: x is the leftmost grandchild of the deepest violation (a)
  PERFORM A ZIG



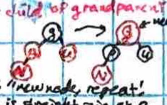
- case 4: x is rightmost grandchild of the deepest AVL violation ⇒ ZAG (mirror of case 1)
- case 2: x is the left right grandchild of the deepest AVL violation
  RIGHT-COMBINED DOUBLE ROTATION (ZAG then ZIG up a level)



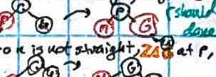
- case 3: x is the right left grandchild of the deepest AVL violation
  MIRROR CASE 2, LEFT COMBINED DOUBLE ROTATION
- Lazy delete: just mark node as deleted, can be overwritten on insert

Red-Black Trees: a BST where:
1. Every node is either red or black
2. Every leaf (NULL leaves) is black and the root is black
3. If a node is red, both children are black
4. Every simple path from any node to its descendant leaves has the same # of black nodes - this is the black height (bh) of nodes

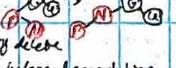
- Space overhead: one bit per node to store color
- Max height: a red black tree w/ n nodes has a height of at most $2\lg(n+1)$ so $\boxed{O(\log n)}$
  - a subtree rooted at x has at least $2^{bh(x)}-1$ internal nodes
  - at least half of the nodes from x to a leaf are black
- INSERT: (Runtime $\Theta(\log n)$) BST insert a RED node, then fix RB properties
  - Assume: N is new node, P = P[N] → N's parent, G = P[P[N]], N's grandparent
  - case 1: N is root → PAINT BLACK
  - case 2: P is black → DONE!
  - Assume: p = left[G], parent is left child of grandparent; N's uncle u = right[G]
  - case 3: else P and U are red
    → color up, black, G red
    → move violation up, make G "new node", repeat!



  - case 4: ELSE if path from G to n is straight, zig at G
    color p black, G red



  - case 5: If path from G to n is not straight, ZAG at P, go to case 4



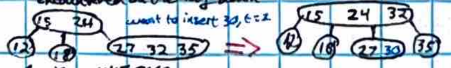
  ★ Other cases symmetric
- Delete: lazy delete, like AVL lazy delete

B-Trees: Non-rotation based balanced search tree
- motivation: minimize # of disk operations or network operations for large databases that don't fit in memory
- Definition: rooted tree w/ following properties
  1) Every node has the following fields:
     n[x] - num keys at node x
     ki[x] - keys in non-decreasing order (linked list/array)
     leaf[x] - true if x is a leaf
  2) If x is an internal node, it also has pointers to its children: $c_1[x]$.. $c_{n+1}[x]$
  3) keys separate children: $c_1[x] \leq k_1[x] \leq c_2[x] \ldots \leq k_n[x] \leq c_{n+1}[x]$
  4) Every leaf has the same depth h
  5) For a fixed system parameter t ≥ 2 min degree:
     - every node other than the root has at least t-1 keys, t children
     - every node has at most 2t-1 keys, 2t children
     - A FULL node has 2t-1 keys

THEOREM: the height of an n-key B-tree is $h \leq \log_t \frac{n+1}{2}$

SEARCH: Search through keys until you find a key > target, go to left child
   Runtime: if LL used: $\Theta(t\log_t n)$

Insert: search for node that should contain new key
   If node has room, add key, finish
   If full, split node, send median to parent, split any full node encountered on the way down



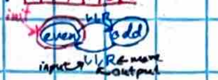
Runtime: LIKE SEARCH
Delete would require MERGING

Turing Machine

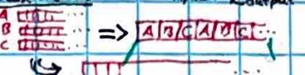
- a hypothetical model that manipulates symbols on a strip of tape according to a set of rules; useful in understanding limits of computation
- Setup: - semi-finite tape

  - initially n leftmost cells full, rest contain blank "b"
  - read/write head (starts at leftmost position)
  - state machine
    - initial state, set of finite states, some non-finite states
- Move: - read symbol from tape pointed to by head
  - change state (using symbol as input)
  - print new symbol on tape, replacing what was there
  - move left/right
  - halts when no move is possible (e.g. we reach blank)
    - if we reach finite state, ACCEPT INPUT, else REJECT

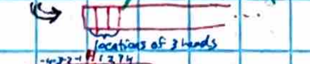
e.g. A machine that accepts an even number of 1's only (assume only 1's as input)
Assume we get the following input:
ENDS IN ODD STATE,
REJECTS INPUT



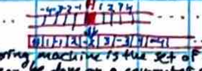
Turing Machine Extensions
- Multi-tapes:



- Multi-tapes & multihead:



- infinite tape:



- language of Turing machine is the set of inputs it accepts
- anything that can be done on a computer can be done w/ a Turing machine
- problems that can't be solved w/ a Turing machine are UNDECIDEABLE
- Halting Problem: determining if a program runs forever is IMPOSSIBLE
- A universal Turing machine accepts a representation of another Turing machine & its input as input (contradiction putting Halts in halts inverter)
- Non-deterministic Turing Machine: could have multiple next state options for a given state/input combo
  - if any of the possible sequence combos end in an accepting state, input is accepted, else REJECTED
  - can solve the same problems as deterministic problems
  - categorize problems
- You can convert from NDTM to DTM, but requires exponential # of states

P and NP:
- goal: solve decision problems - yes/no answer
  - e.g. "Find shortest path from s to u" - not a decision problem
    instead, "is there a path from s to u w/ length k" - a decision problem
- P: set of languages that give a result in polynomial time when run on a deterministic TM
- NP: set of languages that can be run in poly time on an NDTM

e.g. P decision problem: Language: all graphs that are 2-colorable
   Input: graph
   output: accept/reject
   Deterministic TM: run BFS, alt colors for nodes

e.g. NP decision problem:
   Language: strings representing graphs w/ a Hamiltonian cycle
   NDTM: check all possible paths in a graph, accept iff any are Hamiltonian

- We know P ⊂ NP ... Does P=NP (probs not)
- NP-complete → hardest problems in NP

Coding Syntax

Classes & Operator Overloading

numCell.h
```
#ifndef NUMCELL_H
#define NUMCELL_H
class numCell{
   private:
      int storedValue;
   public:
      numCell();
      numCell(int);
      int read();
      void write(int);
      numCell & swap(numCell &);
      numCell & operator=(numCell);
      numCell operator+(numCell &);
};
#endif
```

numCell.cpp
```
#include "numCell.h"
numCell::numCell()
{ this->storedValue=0; }
numCell::numCell(int val)
{ this->storedValue=val; }
int numCell::read()
{ return this->storedValue; }
void numCell::write(int val)
{ this->storedValue=val; }
numCell & numCell::swap(numCell & cell)
{
   int temp=this->storedValue;
   this->storedValue=cell.read();
   cell.write(temp);
}
numCell & numCell::operator=(numCell cell)
{ write(cell.read()); }
numCell numCell::operator+(numCell & cell)
{ ~~[illegible]~~
   numCell newCell(this->storedValue()+cell.read
   return newCell;
}
```

main.cpp
```
#include <iostream>
#include "numCell.h"
using namespace std;
int main()
{ numCell num1;
  numCell num2(15);
  num1.write(30);
  cout<<num1.read()<<endl;
  num1.swap(num2);
  numCell num3;
  num3=num1+num2;
  return 0;
}
```

Vectors (#include <vector>)
```
vector<long> prods;
prods.push_back(5);
v.size()
v[i]
```

Read From File:
```
#include <fstream>
maxProd(string fname)
{
   ifstream input;
   input.open(fname.c_str());
   while(input>>num)
   { cout<<num; }
}
```

Get Args From Command Line
```
int main(int argc, char * argv[])
{
   if(argc<2)
   {
      cerr<<"please input a filename"<<endl;
      return 1;
   }
   string fname(argv[1]);   argv[0]=prog name
   //do stuff
   return 0;
}
```

Lists & Iterators (same class)

main.cpp
```
#include <limits.h>  (#include "numcell.h")
#include <list>
#include<iterator>
int main()
{ const int ARRAY_LENGTH=5;
  int a[]={10,15,2,30,7};
  list<numCell> l;
  for(int i=0;i<ARRAY_LENGTH;i++)
  {
     l.push_back(numCell(a[i]));
  }
  int min=INT_MAX,
      sum=0;
  list<numCell>::iterator it;
  for(it=l.begin();it!=l.end();++it)
  { int curVal=(*it).read();
    sum+=curVal;
    min=(curVal<min)?curVal:min;
  }
  int sum_no_min=sum-min;
  return 0;
}
```

SORTING

Merge Sort: Divide & conquer

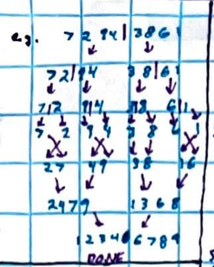
```
[Mergesort(A[1... ⌊n/2⌋])
[Mergesort(A[⌊n/2⌋+1...n])
[Merge halves
```

Runtime: $T(n) = 2T(n/2) + n$
$\Theta(n\log n)$ ALL CASES

Space: $\Theta(n)$

Advantages: Faster!
Disadvantages: not constant space
bad for smaller

e.g.



Quicksort: Randomized Divide & Conquer

1. Select pivot at random or rightmost element
2. Place all elements ≤ pivot to left of pivot
3. Place all elements > pivot right of pivot
4. Quicksort(left of pivot partition)
5. Quicksort(right of pivot partition)

Space: can be done w/ no extra space (constant)
Runtime: $\Theta(n\log n)$ avg & best
$\Theta(n^2)$ worst case

e.g.
10 15 2 1 12 3
2 1 3 10 12 15
1 2 3 10 12 15
1 2 3 10 12 15

Counting Sort

ASSUMPTION: n inputs, each in range [0, m), all ∈ ℤ
e.g. 3 5 4 1 3 4 1 4
- create an array B of size m, init. to 0
- for each occurrence of x in A, B[x]++

```
B: 0 1 2 3 4 5
   0 0 0 0 0 0
   1     1 1 1
   2     2 2
           3
```

- spit out index i of B, B[i] times:

RUNTIME: $\Theta(n+m) \to \Theta(\max(m,n))$
SPACE: IF stable, $\Theta(m+n)$

FASTER
NOT VERSATILE!

*BUBBLE SORT
GOOD FOR SMALL/
ALMOST SORTED
INPUTS

Radix Sort

ASSUMPTION: n inputs, all k digits long, each digit in range [0, d) ← base

START W/ RIGHTMOST DIGIT

```
input A, k
  for i=1 k
    use a stable sort (counting sort) to sort array on digit i
```

Runtime: $\Theta(k(n+d))$ → counting sort k times!
- Space: $\Theta(n)$ → one entire array

Can be used by any base, chars, etc

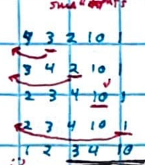
Insertion Sort

Place each value in its correct location as you reach it, shift everything else over.

```
For i = 0 to |A|-1
value = A[i]
j = i-1
while j≥0 and A[j] > value
  A[j+1] = A[j]
  j--
A[j+1] = value
```

Runtime: worst case $\Theta(n^2)$
best case $\Theta(n)$
Space: sort in place (constant)

and small inputs → Advantages: constant space, faster for best case
Disadvantages: slow for non-small inputs



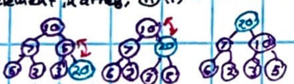
Heap Sort

- heap: a complete binary tree w/ max-heap or min-heap property

0	1	2	3	4	5	6
x	10	7	6	2	1	5

- heap can be stored in an array
- children of i: 2i, 2i+1, parent of i: ⌊i/2⌋
- Find max: Go to first element in array, $\Theta(1)$
- insert:

```
heap_insert(A, key)
    heapsize(A)++
    i ← heap-size(A)
    A[i] ← key
    while (i > 1 and A[parent(i)] < A[i])
        swap (A[i], A[parent(i)])
        // BUBBLE UP
        i ← parent i
```

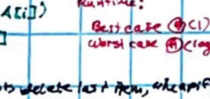


Runtime:
Best case $\Theta(1)$
Worst case $\Theta(\log n)$

- Extract max: swap last item w/ root, delete last item, "heapify"

```
heapify(A, i) ← BUBBLE DOWN!
    If A[i] ≥ max (left(i), right(i))
        nothing
    else
        swap (A[i], max_child)
        Heapify(A, i of maxchild)
```

Delete 20:



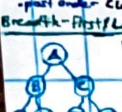
Runtime: $\Theta(\log n)$ worst case
(Heapify + const)

- SORTING W/ A HEAP: insert n elements & extract max n times OR build heap from a random array ($\Theta(n)$) and extract max n times
Runtime: $\Theta(n\log n)$ both cases
- Increase key: increase val of i, heapify $\Theta(\log n)$ worst case
- Delete: swap w/ last elem, delete heapify, $\Theta(\log n)$
bubble down
- Build heap from random array: Heapify from right ⌊|A|/2⌋ to 1, skip leaves, worst case $\Theta(n)$

TREE TRAVERSAL

Depth-First:
- inorder: (left, root, right) D B E A F C G
- preorder: (root, left, right) A B D E C F G
- post order (left, right, root) D E B F G C A

Breadth-First / Level order: by level, left to right A B C D E F G



DATA STRUCTURES

Array: continuous set of containers. Better cache locality
- Access time: $\Theta(1)$ MORE MEM EFFICIENT
- Insert time: $\Theta(1)$ if space available,
else allocate new array, copy all elems, insert new elem $\Theta(n)$ MAINTAINING sorted

Linked list: non-continuous containing a collection of unordered linked elements called nodes
NOT FIXED SIZE!
- Access time: $\Theta(n)$
- Insert time: $\Theta(1)$, $\Theta(n)$ if you include finding position

ADVANTAGES: DYNAMIC SIZE, ease of insertion/deletion
DISADVANTAGES: no random access (can't do binary search), extra memory for pointer)

Stack: LIFO Data structure
- push: add to tail $\Theta(1)$
- pop: remove from tail $\Theta(1)$
- implement w/ array: just need tail pointer

Priority Queue: all elements assigned a priority, element w/ highest priority dequeued first
- insert(item, priority) $\Theta(\log n)$
- getHighestPriority() $\Theta(1)$
- deleteHighestPriority() $\Theta(\log n)$

*Implement using heap!

Hashing m = table size, k = key, frac() = fractional portion
- division hash: m a large prime, hash val for k is k%m
  - uniformly random numbers will map uniformly to table
- multiplication hash: h(k) = trunc(m * frac(k*A)), 0<A<1
  - slower than div. hash, both simple uniform
- COLLISION RESOLUTION:
  - open hashing: each item in table a linked list
    - insert: insert to head $\Theta(1)$, a perfect dist. $\Theta(n/m)$
    - search/delete: worst case all elems in one place, $\Theta(n)$
    - LOAD FACTOR: of a hash, α = n/m, average length of each chain
    - THEOREM: simple uniform hash tables w/ chain collision res. take $\Theta(\alpha+1)$ to search, try to use α ≤ 1
  - closed hashing: (open-addressed hash)
    - each slot holds an element, α cannot exceed 1, ideally α<.5
    - probing sequence: if hash position free, place item, else go to next position in sequence
    - Linear probing: $h_i(k) = [h'(k) + i] \bmod m$, $i = 0 \dots m-1$ ($h'(k)$ given)
    PROBLEM: primary clustering, interferes w/ neighboring hashes, results in long sequence of probes
    - Quadratic probing: $h_i(k) = [h'(k) + c_0 + c_1 i + c_2 i^2] \bmod m$
    Problem: secondary clustering, interference b/w items w/ same initial hash, could get unused slots
    - Double Hashing: use second ary hash for probing sequence
    $H_i(k) = (h_1(k) + i h(k)) \bmod m$
      - solves secondary clustering, larger comp. overhead
    - search: apply hash, search through probe sequence until you find an empty slot
    - delete: LAZY DELETE: don't remove, mark deleted
- Rehashing: periodically creating a larger hash table & reinserting elems

TREES

Binary Search Trees: all nodes in left subtree of x are ≤ x, all nodes in right are ≥ x
- trees: one root node, nodes have children nodes, all nodes except root have one parent
- ACYCLIC & FULLY CONNECTED
- Binary tree: every node has 0, 1, or 2 children
- Complete Binary tree: all levels have max # of nodes except bottom. Fill L → R
- FULL Binary tree: each node has 2 children OR is a leaf
  - THEOREM: # of leaves = # internal nodes +1 for full binary trees
- A tree w/ n nodes has n-1 edges
- search(x, val) x a pointer (to root initially)

```
if (x == null)
  return x
else if key(x) > value
  search(left(x), value)
else
  search(right(x), value)
```

Runtime: $\Theta(\text{height})$
- worst: $\Theta(n)$
- best $\Theta(\log n)$

- Find max: go right until you run out of nodes
- Find min: go left until you run out of nodes
- Insert: search to find where to insert, insert where we fell off
- Successor: next largest element after v in tree

```
if there is a right subtree
    return its min
else
    go up until you get to a node w/ a right parent
```

$\Theta(\text{height})$

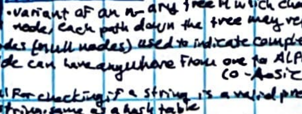
- Delete:

```
if v a leaf, delete
if v has one child, connect v's parent to v's child
else
    swap v w/ successor v
    delete v in its new location
```

$\Theta(n)$ worst
$\Theta(h)$ on avg.

AVL Tree: a balanced BST where for every internal node v, the height of the left & right subtrees differ by one at MOST
- height = distance from farthest leaves, max height = log(n)
- insertion: same as BST, fix AVL violations (still $\Theta(\log n)$ since rotations are constant time

ROTATIONS



- CASES: suppose we insert a node to a valid AVL tree
- case 1: x is leftmost grandchild of deepest violation ZIG
- case 2: x the left-right grandchild of the deepest AVL violation
RIGHT-COMBINED DOUBLE ROTATION (ZAG THEN ZIG UP A LEVEL)
- case 3: x is the right-left grandchild of the deepest AVL violation
MIRROR CASE 2, LEFT-COMBINED DOUBLE ROTATION
- case 4: x is rightmost grandchild of deepest AVL violation ZAG
- Delete: Lazy, just mark nodes as deleted, overwrite on insert

Trie (Prefix Tree)
- variant of an n-ary tree in which characters are stored @ each node, each path down the tree may represent a word
- * nodes (null nodes) used to indicate complete words
- a node can have anywhere from one to ALPHABET-SIZE +1 children (0 - A-SIZE)
- useful for checking if a string is a valid prefix in O(k) time, k = length of string, same as a hash table

- Rearranging letters in a word:
$\frac{{}_nP_k}{x_1! \dots x_p!}$
n = word length
k = new subset length
$x_1 \dots x_p$ = # of times letter i repeats

- Binomial Theorem:
$(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^{n-i} y^i$

Big O: $f \in O(g)$ if f grows no faster than g, $\lim_{x\to\infty} \frac{f(n)}{g(n)} \neq \infty$

$\Theta(g)$ the same rate as $\neq \infty, 0$

GRAPHS

- directed graph: G(V,E) such that V is a set of vertices and E ⊆ V×V is a set of edges
  - each edge an ORDERED set of vertices (v,w) where v,w ∈ V
- undirected graph: G(V,E) s.t. V a set of vertices, E a set of edges, each edge UNORDERED pair of vertices
- degree: of a node = # of edges directly connected to a node
  - in degree # pointed in, out-degree # pointed out
- adjacent vertices: vertices connected by an edge
- weighted graph: graph w/ weight labels on edges

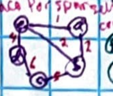
GRAPH REPRESENTATIONS

- adjacency list: table of vertex w/ its neighbors in a sorted list/array of sorted linked lists/arrays

vertex	neighbors
a	be
b	aec
c	bd
d	ce
e	abd

- space: $\Theta(V+E)$
- adjacency time: sorted: $\Theta(\log V)$
unsorted: $\Theta(V)$
- add edge: Sorted: $\Theta(V)$
unsorted: $\Theta(1)$

saves space for sparsely connected



(for directed graphs, list neighbors w/ outgoing edges only)

- adjacency matrix: Rows & columns are vertex names for source & destination, contains 1 for an edge, 0 for no edge

	a	b	c	d	e
a	0	1	0	0	1
b	1	0	1	0	1
c	0	1	0	1	0
d	0	0	1	0	1
e	1	0	1	1	0

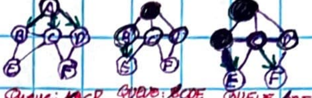
- space: $\Theta(V^2)$
- adjacency time: $\Theta(1)$
- add edge: $\Theta(1)$

- For directed, 1 only for directed edge (where row = source)
- For weighted graph, replace 1 w/ weights, 0 w/ null

- edge list: array/list/vector of edges & overall # of vertices
{{a,b}, {b,c}, {d,c}, {d,e}, {c,a}, {b,e}}
- space: $\Theta(E)$
- add edge $\Theta(1)$
- adjacency time: $\Theta(E)$

A tree has one path b/w any two vertices

GRAPH TRAVERSAL:
- BREADTH-FIRST SEARCH: start w/ a node & visit its neighbors, their neighbors, etc.
- data structure: graph & a queue
  - 3 colors, white = unvisited, gray = working on, black = done w/
- Runtime: $\Theta(V+E)$ by aggregate analysis



SHORTEST PATH (# of EDGES)
NOW EMPTY QUEUE!

```
BFS(G,S)
  color all vertices white, S gray
  d(v) = ∞, d(s) = 0, π(v) = null
  enqueue S
  while queue is not empty
    h head of queue
    for each neighbor n of h,
      if n = white
        color gray
        enqueue n
        d(n) = d(h)+1
        π(n) = h
    dequeue(h)
    color h black
```

SPANNING TREE
Depth-First Search: for each node, go as far as possible in one direction, backtrack & restart
- data structure: STACK (recursion), node coloring again
- Runtime: $\Theta(V+E)$ agg. analysis

```
DFS_visit(G,A)
  DFS_visit(G,B)
    DFS_visit(G,C)
      DFS_visit(G,D)
      done
      DFS_visit(G,E)
      done
    done
  done
done
```

```
DFS(G)
  color all vertices white
  π(v) = null, time = 0
  for each white vertex v
    DFS_visit(G,V)
DFS_visit(G,V)
  color V gray
  for each node n adjacent to V
    if n is white
      π(n) = V
      DFS visit(G,n)
  color V black
  F[V] = time++
```

VISIT ALL
*Works on unconnected graphs!

SPANNING TREE
TOPOLOGICAL SORT (reverse finish time order)
↳ linear ordering of vertices in a graph
CAN BE USED FOR CYCLE DETECTION!

TURING MACHINES & NP COMPLETENESS

- TURING MACHINES: a hypothetical model that manipulates symbols on a piece of tape according to a set of rules, useful in understanding limits of computation
- Setup: semi-infinite tape
  - initially, n leftmost cells full, rest blank "B"
  - read/write head starts @ leftmost position
  - state machine: initial state, set of finite states, some non-finite states
- move:
  - read symbol from tape pointed to by head
  - change state
  - print new symbol on tape replacing what was there
  - move left/right
  - halts when no move is possible (reach a BLANK)
    - if we reach finite state, ACCEPT input, else REJECT
- Everything that can be done on a computer can be done w/ a Turing machine
- Problems that can't be solved w/ a Turing machine are UNDECIDEABLE
- Halting problem: determining if a program runs forever is IMPOSSIBLE
- A universal TM accepts a representation of another Turing machine & its input as input
- Non-deterministic TM: can have multiple state options for a given state/input combo
  - if any of the possible sequence combos end in an accepting state, input is accepted, else rejected
  - can solve same problems as deterministic TMs
  - categorize problems
- You can convert from NDTM to DTM, but requires exponential # of states

P⊂NP... NP⊂P ?

- P & NP:
  - goal: solve decision problems (yes/no answer)
  e.g. Is there a path from s to v w/ length k,
  NOT what is the shortest path from s to v
  is a graph two-colorable?
  - P: problems that can be solved by a DTM in poly time
  - NP: problems that can be solved by an NDTM in poly time
  solution quickly verifiable
  - NP-complete: hardest problems in NP ↳ strings representing graphs w/ no Ham. cycle
    - no polynomial time algorithm has been discovered, but no one can prove there isn't one

EXAMPLE NP-COMPLETE:
- Boolean Satisfiability: is there an interpretation of the variables of a boolean formula s.t. the value is TRUE
$a\bar{b}$ → satisfiable, a = true, b = false
- Knapsack: given a set of items, each w/ weight & value, determine # of each item to include to be <= a weight limit & have highest total value
- Traveling salesman: given a list of cities & the distances b/w each pair, determine the shortest possible route that visits each city & returns to its origin city?
- Hamiltonian Path: does a Hamiltonian path exist in a given graph? → special case of travelling salesman

DISCRETE REVIEW

- permutation: number of ORDERED subsets of k out of n elements w/ NO REPETITION: ${}_nP_k = \frac{n!}{(n-k)!}$
- combination: number of UNORDERED subsets of k out of n elements w/ NO REPETITION: ${}_nC_k = \binom{n}{k} = \frac{n!}{(n-k)!k!}$
- number of ORDERED SUBSETS of k out of n elements w/ REPETITION $= n^k$
- number of UNORDERED SUBSET of k out of n elements w/ REPETITION
$= \binom{n+k-1}{k} = \frac{(n+k-1)!}{(k!)(n-1)!}$