Algorithm – Recipe for solving a problem
non-ambiguous for solving a problem

Important issues for algorithms
correctness
efficiency
choice of Data Structures

ADT- abstract data type

eg: coin changing algorithm (greedy method) ← seen this before

Everyday Coin Changing

Make change of C cents using **fewest** coins

Quarters = C / 25;
R = C – 25*Quarters;
Dimes = R / 10;
R = R – 10*Dimes;
Nickels = R / 5;
Pennies = R – 5*Nickels

PSN. Is this algorithm still correct, i.e., are fewest coins used?
(pause video to think about this)

PSN 1/12

this algorithm is not correct for the example of ~~35¢~~ 30¢
this gives you
1 Quarter
1 Dimes
9 Pennies
with a total of 11 coins without nickles

bc 3 Dimes
vs
1 quarter
5 pennies

Proof by contradiction (apply a proof to say it works but when it is this it does not work)

PSN. Obtain upper bounds p_{10}, p_5, p_1
(pause video to think about this)

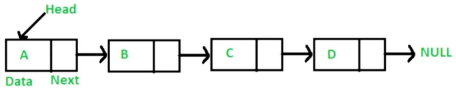
the upper bounds of p_{10}, p_5, p_1 is the max number of coins you can have

So
$P_{10} = 2$
$P_5 = 1$
$P_1 = 4$
} according to the greedy method

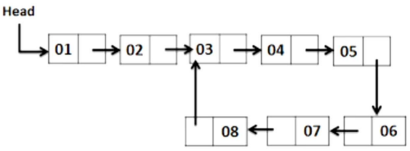
Coin changing algorithm is NP Hard

PSN: Broken Linked List

Give an efficient algorithm that uses a minimum of extra space for testing whether a linked list is "broken", i.e., has a next pointer that points to previous node in the linked list, thereby forming a cycle.



Not broken



Broken

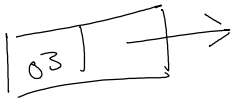
Hare and Tortise algorithm

```
Hare = Head;
Tortoise = Head;
Broken = false;
while (Hare != NULL && Broken == false) {
      Hare = Hare -> Next;
      if (Hare == Tortoise) Broken = true;
      if (Hare != NULL) {
            Hare = Hare -> Next;
            if (Hare == Tortoise) Broken = true;
      } //END IF
      Tortoise = Tortoise -> Next;
} //END WHILE
```

if you could store the pointers in a list as you go though the list then if any pointer you see is already in the list then you know it loops

eg: if  and

 both point to



then it loops

Recursion is a powerful tool for algorithms (Recursive functions)
eg: Devide and conquere, Dynamic programing, Backtracking

Rewsion in a computer is implemented as a stack

Devide and Conquve - specicl ccsc of recursion

merge sort!

Backtracking and Branch and Bound

↓ ↓

Depth first search and breath first search

Powers

Compute X^p for a given positive integer exponent p

we want the fastest algorithm for computing powers

a looping algorithm to calculate this number is 2^n

```
function Powers(x,p) recursive
  Input: x (a real number), p (a positive integer)
  Output: x^p
   if p = 1 then return (x)
   if even(p) then
      return(Powers(x*x,p/2))
   else
      return(x*Powers(x*x,(p – 1)/2))
   endif
end Powers
```

this is a better algorithm

Crecursive calls

$2 \log_2 P$ multiplications

Problem Solving Notebook: Number of digits

Show that the number n of decimal digits of p is approximately $\log_{10} p$.

What about number of binary digits? Octal digits? hexadecimal digits?

Proof by example?

if $P = 1300$ so $n = 4$

so $4 \approx \log_{10} 1300 = 3.1139$

$4 \approx \log_{10} 9999 = 3.999$

$1300_{10} \rightarrow$ (base 2) $= 10\,1000\,10100$

$n = 11$

$11 \approx \log_2 1300 = 10.344$

$1300_{10} \rightarrow 2424_8$

$n = 4$

$4 \approx \log_8 (1300) = 3.448$

$1300_{10} \rightarrow 514_{16}$

$n = 3$

$3 \approx \log_{16} (1300) = 2.586$

What about using simpler formula:

$(x*x)^{p/2} = x^p$ if p is even, $x*x^{p-1} = x^p$ if p is odd

Is it still efficient?

```
function Powers(x,p) recursive
  Input: x (a real number), p (a positive integer)
  Output: x^p
 if p = 1 then return (x)
 if even(p) then
     return(Powers(x*x,p/2))
 else
     return(x*Powers(x,p – 1)
 endif
end Powers
```

13

there are more recursive calls here than in the example above

non recursive example

1. Compute the binary representation of *p*
2. Initialize *Pow* to 1
3. Scan from left-to-right starting with **second** position
   - 3.1 **if** 0 is encountered square *Pow*
   - **else** // if 1 is encountered
     - square *Pow* and multiply by *x*
   - **endif**

Problem-Solving Notebook
Changing to Binary

Give pseudocode for a recursive function *BinRep*(*n*) for converting an number *n* to its binary (base 2) representation stored in a string. Assume + performs the operation of adding a digit, i.e.,

"10001101" + 0 → "100011010"

The strategy for this is to get to a binary number from a decimal number
The easiest way to do this is to divide by 2 throughout and see where the remainders are

Eg n = 13 = 1101
13/2 = 6 R 1
6/2 = 3 R 0
3/2 = 1 R 1
1/2 = 0 R 1

Adding all the remainders together you get 1101 which gets you your binary number

```
Function DectoBin(number)

Remainder = number mod 2
newNumber = number / 2
If newNumber > 0:
    nextDigit = DectoBin(newNumber)
Else
    Return remainder;
Nextdidgit = next digit + Remainder
Return nextDigit
```

more of a 2 way recursive

---

atoms in the universe 10^{83} ← modulo exponentiation

to do the method $x^{(10^{83})}$ would require more disk than any other computer

to solve this we have to use [modulo some integer k]

this makes it so the number never exceeds

$(x-1)^2$

$xy \bmod k = (x \bmod k)(y \bmod k) \bmod k$

or in $y=x$

$x^2 \bmod k = (x \bmod k)(x \bmod k) \bmod k$

```
function ModularExponentiation(x,p,k) recursive
  Input: x, p, k (positive integers)
  Output: x^p (mod k)
    if n = 1 then return (x mod k)
    if (n mod 2 = 0) then
       return ( Powers(x*x mod k,p/2))  mod k )
    else
       return ( x*Powers(x*x mod k,(p – 1)/2)) mod k )
    endif
end Powers
```

Converse is a hard problem

given x, p and $x^n \bmod p$ there is no known efficient algorithm to compute n

Seminal Example from Cryptography cont'd

- First Alice and Bob communicate with each other to decide on the value of a integer base b and a large integer p.
- Now Alice, Bob, and Eve all know b and p.
- Then Alice chooses a large integer n, (which ONLY she knows), and sends Bob the value $b^n \bmod p$.
- Then Bob chooses a large integer m (which ONLY he knows), and sends Eve $b^m \bmod p$.
- Now they ALL know values b, p, $b^n \bmod p$, and $b^m \bmod p$.

26

PSN. How about computing key as follows

$K = (b^m \bmod p)(b^n \bmod p) = b^{m+n} \bmod p$

Since everyone knows all the values everyone can compute K

One person can compute $(b^m \bmod p)^n$

one person can compute $(b^n \bmod p)^m$

with large values of n there is no efficient way to compute n

Discrete Logarithm Problem

GCD - Greatest Comon Devisor
largest inkegers that devides two numbers
You can compute GCD by using prime factorization
$3000 = 2^3 \times 3 \times 5^3$ ← PF

Euclids algorithm is the basis of
GCD(a,b) = GCD(b,r) → this continus until the remainder is 0 and b is your GCD

PSN. Using Euclid's algorithm compute
gcd(585,1035)

gcd(585, 1035)
585 = 0 x 1035 + 585
1035 = 1 x 585 + 450
585 = 1 x 450 + 135
450 = 3 x 135 + 45
135 = 3 x 45 + 0
gcd(585, 1035) = 45

Recursive version of Euclid GCD

```
function EuclidGCD(a,b)
 Input: a, b (nonnegative integers)
 Output: gcd(a,b)
    if (b == 0)
          return a
    else
          r = a mod b
          return EuclidGCD(b,r)
    endif
end EuclidGCD
```

Recursive

Nonrecursive version

```
function EuclidGCD(a,b)
Input: a, b (nonnegative integers)
Output: gcd(a,b)
   while b ≠ 0 do
        Remainder = a mod b
        a = b
        b = Remainder
    endwhile
    return(a)
end EuclidGCD
```

Non - Recursive

most iterations of the algorithm
if a < b a and b get swapped
euclids algorithm takes the lagest for fibbinachi numbers where
a = fib(n) b = fib(n+1)

applications of GCD
Lowest comon Multiple (LCM)
Fraction in lowest form
RSA Cryptographic algorithm

$$LCM(a,b) = \frac{ab}{gcd(a,b)}$$

$$\frac{a}{b} \text{ in simplest form} = \frac{a/gcd(a,b)}{b/gcd(a,b)}$$

2nd vid (evaluating polynomials)

$p(x) = a_0x_0 + a_1x_1 \cdots + a_nx_n$

taylor series for e^x

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots \frac{x^n}{n!}$$

Straightforward solution

function *PolyEvalNaive*(*a*[0:*n*],*v*)
Input: *a*[0:*n*] (an array of real numbers), *v* (a real number)
Output: the value of the polynomial $a_nx^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$ at $x = v$

```
Sum ← a[0]
Product ← 1
for i ← 1 to n do
    Sum ← Sum + a[i] * Powers(v,i)
endfor
return(Sum)
end PolyEval
```

Monday, January 24, 2022 3:34 PM

Computing time Complexity
function of the input size $O(n)$ ← depends on application size

Measuring the complexity (Basic operation)
count how many times a Basic operation is performed

note: different inputs may perform differently
(not just change in n)

Orginize in
(fewest) Best Case ⟶ $B(n) = \min\{ t(I) \mid I \text{ in } D_n \}$
(most) Worst Case ⟶ $W(n) = \max\{ t(I) \mid I \text{ in } D_n \}$
Average case

worst case is usely is more important than any other case

↓

Usely the expected value
Complexity with any random input
dependent on probability distribution

$$A(n) = \sum_{i=B(n)}^{W(n)} i\, p_i$$

p_i : number of basic operations

Linear Search Complexity ⟶ base op (comparsion of list element to search element)

Best Case Complexity = $O(1)$
Worst Case Complexity = $O(n)$

Average case Complexity = $p_i = \frac{1}{n}$

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

Binary Search

Wost Case Complexity = $\lceil \log_2(n+1) \rceil \approx \lceil \log_2 n \rceil$
Best Case Complexity = 1

PSN 1/21

Give C++ code for the a recursive version of binary search where there is no equality check with midpoint is made and analyze your algorithm

```
{
  int Mid
  if(low > high)
     return -1
  if (first == last) {
     if (List[low] == SearchElement) return low
        else return -1;

  mid = (low + high)/2
  if (search element < list[mid] < 0
       return binarySearch(List, SearchElement, low, mid)
  else
     return binarySearch( list, SearchElement, mid+1 high)
}
```

if $n = 2^k$

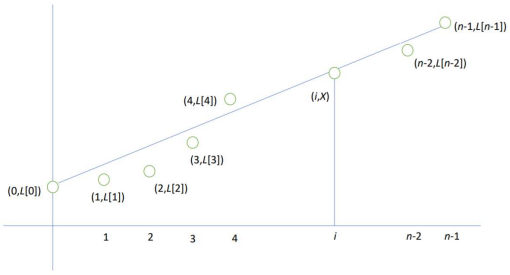
Best Case $= B(n) = k = \log_2 n$

Worst Case $= W(n) = k = \log_2 n$

Average Case $= A(n) = \log_2 n$

Interpolation Search – like a binary search but values in the original list L[0:n-1] but align in a straight line – (same increment)

Interpolation Search



Recursive implementation – low and high and perform the interpolation search on the sublist L[low, high]

PSN. Obtain a formula for *i* in terms of *X*, *low*, *high*, *L*[*low*], *L*[*high*]

$\Delta y / \Delta x = (L(High) - L(low)) / (high - low)$
$\Delta y / \Delta x = (X - L(low)) / (i - low)$
$i = low + (X - L(low))(high - low) / (L(high) - L(low))$

PSN. Write pseudocode for recursive version of Interpolation Search.

```
if high < low then return (-1)
if X < L[low] or X > L(high) return(-1)
  mid ← low + (X - L(low))(high - low) / (L(high) - L(low))
if X = L[mid] return mid
if X < L(mid) then
   interpolation search(L[0:n-1], low, mid - 1, X)
else
   interpolation Search ( L[0 : n-1], mid + 1, high, X )
end if
```

Worst case complexity of interpolation Search = $W(n) = n + 2$
average case complexity of interpolation Search = $A(n) = \log_2(\log_2 n)$

Computing the maximum element in as list

```
function Max (L[0:n – 1])
Input:  L[0:n – 1] (a list of size n)
Output:        returns the maximum value occurring in L[0:n – 1]
  MaxValue ← L[0]
  for i ← 1 to n – 1  do
       if L[i] > MaxValue then
            MaxValue ← L[i]                  //update MaxValue
       endif
  endfor
  return(MaxValue)
```

Analysis of Max and Min
$B(n) = A(n) = W(n) = n - 1$

Min can be implemented in similar ways

```
        endif
    endfor
    return(MaxValue)
end Max
```

$B(n) = n - 1$

$W(n) = 2n - 2$

Pseudocode for *MaxMin3*

```
procedure MaxMin3(L[0:n – 1],MaxValue,MinValue)
Input: L[0:n – 1] (a list of size n)
Output:    MaxValue,MinValue (the maximum and minimum values in L[0:n – 1])
   if even(n) then  //n is even
            MM(L[0],L[1],MaxValue,MinValue)
            for i ← 2 to n –  2 by 2 do
                        MM(L[i],L[i + 1],b,a)
                        if a < MinValue then MinValue ← a endif
                        if b > MaxValue then MaxValue ← b endif
            endfor
   else //n is odd
            MaxValue ← L[0]; MinValue ← L[0];
            for i ← 1 to n – 2 by 2 do
                        MM(L[i],L[i + 1],b,a)
                        if a < MinValue then MinValue ← a endif
                        if b > MaxValue then MaxValue ← b endif
            endfor
   endif
end MaxMin3
```

Better Algorithm

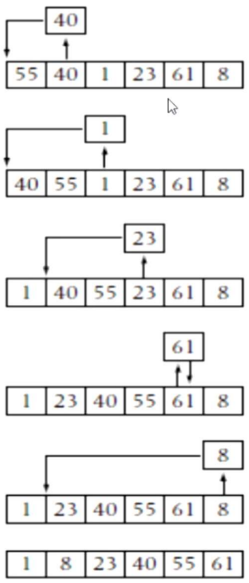
```
procedure MaxMin2(L[0:n – 1],MaxValue,MinValue)
Input:    L[0:n – 1] (a list of size n)
Output:   MaxValue,MinValue (maximum and minimum values occurring in
                                                        L[0:n – 1])
   MaxValue ← L[0]
   MinValue ← L[0]
   for i ← 1 to n – 1  do
            if L[i] > MaxValue then
                        MaxValue ← L[i]
            else
                        if L[i] < MinValue then
                                    MinValue ← L[i]
                        endif
            endif
   endfor
end MaxMin2
```

$B(n) = A(n) = W(n) = \lceil 3n/2 \rceil - 2$

computing time - Insertion sort
Recursive Strategy - Start with L[0]
perform n-1 passes until already in sorted position

Sudo code for Insertion

```
procedure InsertionSort(L[0:n – 1])
Input:    L[0:n – 1] (a list of size n)
Output:   L[0:n – 1] (sorted in increasing order)
      for i ← 1 to n – 1 do //insert L[i] in its proper position in L[0:i – 1]
            Current ← L[i]
            position ← i – 1
          while position ≥ 0 .and. Current < L[position] do
                                                  //Current must precede L[position]
                L[position +1] ← L[position]          //bump up L[position]
                position ← position – 1
          endwhile
        //position + 1 is now the proper position for Current = L[i]
            L[position +1] ← Current
        endfor
end InsertionSort
```



example of insertion sort

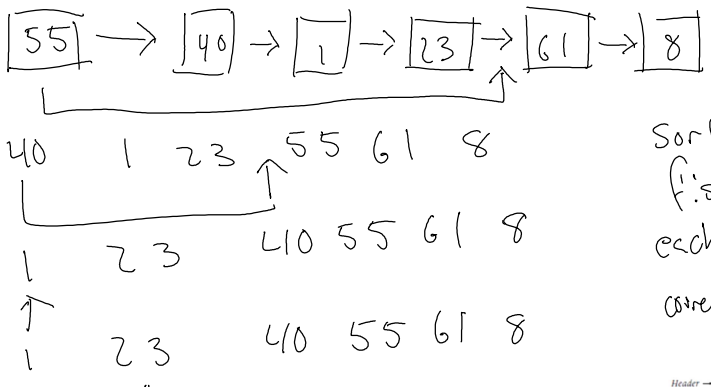
Best case $B(n) = n-1$ (only one comparison for each element)
Worst case $W(n) = \frac{n^2}{2} - \frac{n}{2}$ (reverse order of a sorted list / already sorted)

Average case $A(n) = \sum_{i=1}^{n} \frac{i+1}{2} = \frac{1}{2}\frac{(n+1)(n+2)}{2} - 1 = \frac{n^2}{4} + \frac{3n}{4} - \frac{1}{2}$

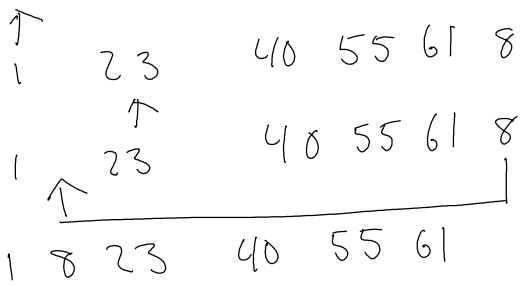
evaluated as $\frac{1}{i}(1+2+\cdots i) = \frac{i+1}{2}$

insertion sort is good for lists that are kind of sorted since it is linear
merge sort is still better
insertion sort is good for a online algorithm (getting data over time)

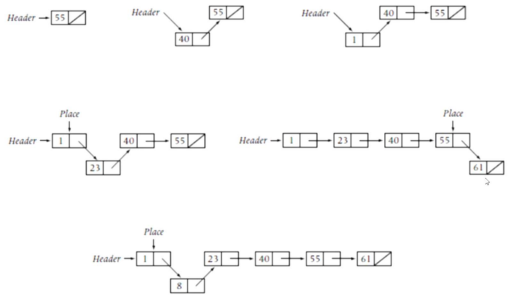
PSN. Show pictorially the action of Insertionsort implemented using a linked list for the same sample list 55 40 1 23 61 8.



Sort in order of the linked list
first store position and hit each element once
correct answer



correct answer



Asymptotic Behavior of functions

$f(n) = 100n + 50$ $g(n) = n^2 - n$

algorithm A has worst-case complexity $f(n)$
algorithm B has worst-case complexity $g(n)$
what is faster?
Algorithm A

Asymptotic Order
$f(n): \mathbb{N} \to \mathbb{R}^+$

Big Oh notation is a approximation

$f(n) \in \mathcal{O}$ exist positive constants c and n_0
$n \geq n_0$

$f(n) \leq cg(n)$ for all $n \geq n_0$

Big Oh $O(g(n))$
Theta $\Theta(g(n))$
Omega $\Omega(g(n))$
Little oh $o(g(n))$
Asymptotically equivalent $\sim g(n)$

PSN. For the example, $f(n) = 100n + 50$ and $g(n) = n^2 - n$, show that

$f(n) \in O(g(n))$.

for $C = 1$ $n_0 = 102$

$100n + 50 \leq 1(n^2 - n)$
$10200 + 50 \leq 1(10404 - 102)$
$10250 \leq 10302$ ✓

Is this definition meaning. For example is

$n^2 - n \in O(100n + 50)$.

Why not just chose a really, really huge constant c? Surely,

$n^2 - n \leq 10000000000000000000000000000(100n + 50)$
for all $n \geq 1$.

PSN. Show this isn't true.

if we divide by n then one side is constant while the otherside goes to infinity
one side $\lim_{n \to \infty} \frac{n^2 - n}{n} = \infty$ $\lim_{n \to \infty} O(100 + 50/n) = 100C$ so as n gets bigger
the n does not work out in the equality

So $100n - 50 \in O(n^2 - n)$ but $n^2 - n \notin O(100n + 50)$

PSN. Show that $\log_a n$ *and* $\log_b n$ *have the same order independent of the choice of bases a and b.*

$\log_a n = (\log_a b) \log_b n$

$\log_a n \in \Theta(\log_b n)$ therefore bases don't matter

PSN. Using the Ratio Limit Theorem, show that worst-case complexity of Binary Search has smaller order than the worst-case complexity of Linear Search, i.e.,

$$O(\log_2 n) \subset O(n).$$

$$\lim_{n\to\infty} \frac{\log_2 n}{n} = \lim_{n\to\infty} \frac{(\log_2 e)\ln n}{n} = \lim_{n\to\infty} \frac{\log_2 e}{n} = 0$$

So $O(\log_2 n) \subset O(n)$

Asymtotic Functions Pt. 2

Proposition Let $P(n)$ be the k^{th} degree polynomial $a_k n^k + \dots a_1 n + a_0$ where $a_k > 0$ then

$P(n) \sim a_k n^k$ $\quad$ $P(n) \in \Theta(n^k)$

PSN. Prove the Proposition.

Since Corollary: $P(n) \in \Theta(n^k)$ then $P(n)$ has the same order as n^k

So there exists positive constraints such that $n \geq n_0$

so $a_k \in C$ and

Possibly $\longrightarrow$ $a_k n^k \leq P(n) \leq a_k n^k$

$$\lim_{n \to \infty} \frac{a_k n^k + \dots + a_1 n + a_0}{a_k n^k} = 1$$

Polinomial Exponential computing times

an algorithm has polynomial computing time if for some pos integer

$W(n) \in O(n^k)$

exponential order $1 < a \leq b$ for some real numbers a and b

$a^n \leq f(n) \leq b^n$

algorithm has exponential computing time if $W(n)$ has exponential order

for any constants k and a where $a > 1$

$O(n^k) \subset O(a^k)$

Order Formula for series

$H(n) = 1 + 1/2 + \dots + 1/n \sim \ln n \in \Theta(\log n)$.

$L(n) = \log(n!) = \log 1 + \log 2 + \dots + \log n \in \Theta(n \log n)$.

$S(n,k) = 1^k + 2^k + \dots + n^k \sim n^{k+1} / (k+1)$

Harmonic Series $H(n)$

$H(n) = 1 + \frac{1}{2} + \dots + \frac{1}{n}$ $\quad$ $H(n) \sim \ln(n)$

PSN. Prove

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

$\Leftrightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

Suppose $f(n) \in \Theta(g(n))$ then $\searrow$

$f(n) \leq c_2 g(n)$, for all $n \geq n_0 \Rightarrow f(n) \in O(g(n))$

$f(n) \le c_1 g(n)$, for all $n \ge n_0 \Rightarrow f(n) \in \Omega(g(n))$

Supose $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$ then

$f(n) \le c_2 g(n)$, for all $n \ge n_0$ and $f(n) \ge c_1 g(n)$, for all $n \ge n_1$,

therefore

$c_1 g(n) \le f(n) \le c_2 g(n) \Rightarrow f(n) \in \Theta(g(n))$

Sum of logs

$L_2(n) = \log_2 1 + \log_2 2 + \cdots + \log_2 n = \log_2 n!$

Prop $L_2(n) \in \Theta(n \log_2 n)$

Proof of $L_2(n) \in O(n \log_2 n)$

$L_2(n) = \log_2 1 + \log_2 2 + \cdots + \log_2 n < \log_2 n + \log_2 n + \cdots + \log_2 n = n \log_2 n$

Proof of $L_2(n) \in \Omega(n \log_2 n)$

$\log_2 1 + \log_2 2 + \cdots + \log_2 n > \log_2 \left(\frac{n}{2} + 1\right) + \log\left(\frac{n}{2} + 2\right) + \cdots + \log_2 n > \log_2 \frac{n}{2} + \log_2 \frac{n}{2} + \cdots + \log_2 \frac{n}{2}$

$= \frac{1}{2} n \log_2 \frac{n}{2} = \frac{1}{2} n \log_2 (n-1) \in \Omega(n \log_2 n)$

Both of these prove that $L_2(n) \in \Theta(n \log_2 n)$

Application of lower bound for comparison sorting algorithms

a comparison-based algorithm for sorting a list is based on comparisons made to elements of the list. They make no prior assumtion about the list other than knowing relative order

For example, the lists (7.2, 1, 8,2), (108, 23.99, 123, 55), (Mary, Ann, Pete, Joe) and (30, π, 31, 12) of size 4 can all be regarded as having the same ordering as the permutation (3,1,4,2). In particular, they each require the same number of comparisons when input to a comparison-based sorting algorithm before they are put into increasing order.

So any list can be represented as a list of n elements in a certain order a sequence of m comparisons will devide the set of $n!$ permutations and at most

2^m classes (Size n)

$m \in \Omega(n \log n)$ $\quad W(n) \in \Omega(n \log n)$

$\Rightarrow$ worst case = $n \log n$ comparison based

Recurrence Relations for Complexity

Binary Search $n = 2^k - 1$ $k = \log_2(n+1)$

uses recursive calls after two comparisons

$W(n) = W((n-1)/2) + 2$ $W(0) = 0$

Setting $t(k) = W(n) = W(2^k - 1)$

$t(k) = t(k-1) + 2$ initial condition $t(0) = 0$

applying repeated substitution

$$\begin{aligned} t(k) &= t(k-1) + 2 = (t(k-2) + 2) + 2 \\ &= t(k-2) + 4 = (t(k-3) + 2) + 4 \\ &= t(k-3) + 6 = (t(k-4) + 2) + 6 \\ &\vdots \qquad\qquad \vdots \\ &= t(0) + 2k = 2k = 2\log_2(n+1) \sim 2\log_2 n \end{aligned}$$

PSN. Obtain recurrence relation for $W(n)$ and solve for the version of Binary Search where no equality check is done until list has size 1. For convenience assume $n = 2^k$.

$k = \log_2 n$

$W(n) = W(n/2) + 1$ Init $W(1) = 1$

applying repeated substitution

$W(n/2) + 1 = (W(n/4) + 1) + 1$

$W(n/4) + 2 = (W(n/8) + 1) + 2$

$\vdots$

$W(n/2^k) + k$

$W(1) + k = 1 + \log_2 n \sim \log_2 n$

Merge Sort vs Quick sort

merge Sort splits the list and sorts smaller elements

best case merge sort uses $n/2$ comparisons and $n-1$ comparisons (worst case)

Recurrence Relation

$W(n) = 2W(n/2) + n$ initial condition $W(1) = 0$

Solving recurrence relations

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

Repeated Substitution

$W(n) = 2W(n/2) + n = 2(W(n/2^2) + n/2) + n$
$= 2^2 W(n/2^2) + 2n = 2^2(2W(n/2^3) + n/2^2) + 2n$
$= 2^3 W(n/2^3) + 3n$

$2^k W(n/2^k) + kn$

by initial condition $W(1) = 0$

PSN. Obtain recurrence relation for the Best-Case Complexity $B(n)$ of Mergesort and solve using repeated substitution.

$B(n) = 2B(n/2) + n/2 = 2(2B(n/2^2) + n/2) + n/2$
$= 4B(n/2^2) + 2(n/2) = 4(2B(n/2^3) + n/4) + 2n$
$= 8B(n/2^3) + 3(n/2)$
$\vdots$
$= 2^k B(n/2^k) + k(n/2)$

$n = 2^k$

$B(n) = 2^k B(1) + k(n/2)$

$B(n) = \frac{1}{2} n \log_2 n$

Average complexity of merge sort

$B(n) \le A(n) \le W(n)$
$\frac{1}{2} n \log_2 n \le A(n) \le n \log_2 n$
$A(n) \in \Theta(n \log n)$

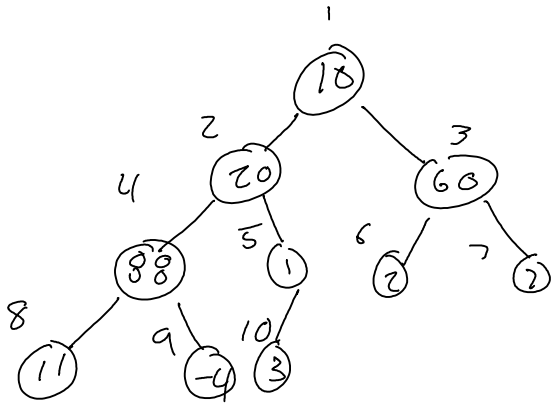
Trees – a way of organizing data
consist of nodes and the top is a parent

PSN. Implementing a full and complete tree

a) Draw complete binary with values on nodes implemented by the array:

0	1	2	3	4	5	6	7	8	9
10	20	60	88	1	2	7	11	-4	3

a)



Given the index i of a node in a complete tree

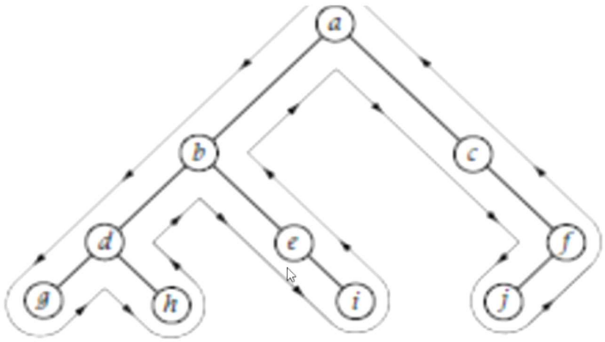
b) Give formula for index of parent of i.

c) Give formulas for indices of left and right children of i.

b) $(i-1)/2$

c) $2*i+1$ and $2*i+2$

Example Binary Tree Traversals



Preorder Traversal: *abdgheicfj*
Inorder Traversal: *gdhbeiacjf*
Postorder Traversal: *ghdiebjfca*

PSN. Code for Binary Tree Traversals

Give C++ code for Preorder, Inorder and Postorder traversals, invoking function Visit() to perform operations on data in the node.

Assume the nodes of the tree are implemented using the structure:

```
typedef int datatype;
struct Node {
    datatype item;
    Node *leftchild;
    Node *rightchild;
}; //END TREENODE

typedef *Node ptrNode
```

```
Void Preorder (ptr Node root)
if (root != null)
  visit (root -> item)
  Preorder (root -> left child)
  preorder (root -> right child)
}
```

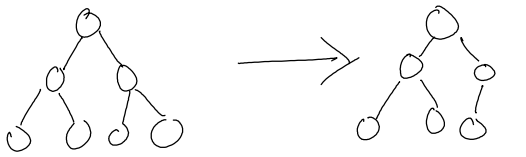
```
Void inorder (ptr Node root)
if (root != null)
    inorder (root -> left child)
    visit (root)
    inorder (root -> right child)
}
```

```
Void Postorder (ptr Node root)
if (root != n
    Postorder (root -> left child)
    Postorder (root -> right child)
    visit (root)
}
```

mathmatical properties of trees
used for analysis of algorithms
Number of edges in a tree
the number of edges of any tree is one less than the number of nodes

PSN. To apply the induction hypothesis, we need to perform an operation that reduces *T* to a tree *T'* having *k* nodes. How to do this?

to do this you would remove a leaf node the deepest node you can have remove the incident edge



$T = k+1$ $T' = k$

Appling induction hypothesis assuming true for $n=k$
any tree with k nodes has $k-1$ edges

$m(T') = n(T') - 1$
$m(T) = n(T') + 1$
$= (n(T')-1)+1$
$= n(T') = n(T) - 1$

Every binary tree with n nodes has depth d at least

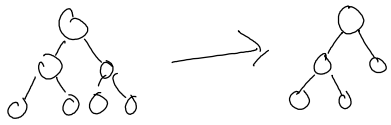
$\lfloor \log_2 n \rfloor$ ← important!

2 tree – every node that is not a leaf has exactly two children
if T is a two tree then leaf nodes = internal nodes + 1
total nodes = 2(internal nodes) + 1
total nodes = 2(leaf nodes) − 1

PSN. To apply the induction hypothesis, we need to perform an operation that reduces T to a tree T' with *k* leaf nodes.

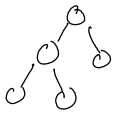
How to do this?

remove 2 leaves? from the tree? to get T' which is T'



this would remove a pair of leaves that have the same parent

PSN. To verify that this construction is valid, we must prove that every 2-tree *T* contains a node, both of whose children are leaf nodes. Proof this result.

lets say $T =$  so $I = 1$ $L = 2$ and $\overline{T} =$  so $\overline{I} = 2$ $\overline{T} = 3$

no matter what the size of the tree is you can always reduce a b tree by removing siblings from the base case all the way to the n case

you can not remove a node and its sibling if the parent has a sibling

---

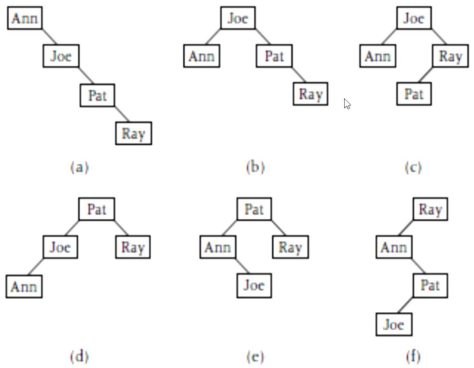
Lower bound of a binary tree in terms of number of leaf nodes

every binary tree with L leaf nodes has depth d at least

$$\lceil \log_2 L \rceil = d$$

Binary Search trees

inorder traversal will output in sorted order



output of inorder traversal

Ann < Joe < Pat < Ray

PSN. Searching a BST

Write a recursive function SearchBST in C++ for search a binary tree for a Search Key. If found it returns pointer to node where found otherwise it returns NULL.

Assume the nodes of the tree are implemented using the structure:

```
typedef int KeyType;
struct Node {
    KeyType Key;
    Node *Left;
    Node *Right ;
}; //END TREENODE
typedef *Node ptrNode;
```

```
Public Node searchBST(node search, keytype key){
    If (search->key < Key){
        //move to the right because key is bigger
        searchBST(search->right, key);
    Else If (head->key > key){
        //the key is less than the left of the list and needs to go smaller
        searchBST(search->left, key);
    Else{
        //the key equals the side of the list it needs return the node
        Return search
    }
}

Solution is kinda the same thing just with root == null check in it
```

Recursive Version SearchBST

```
ptrNode SearchBST(ptrNode Root, KeyType SearchKey)
{   //returns pointer to node that contains search key or
    // returns NULL if SearchKey not found
if (Root == NULL)
     return(NULL);
else
    if (SearchKey == Root -> Key)
        return Root;
    else if (SearchKey < Root -> Key)
            return SearchBST(Root -> Left, SearchKey);
        else
            return SearchBST(Root -> Right, SearchKey);
} //END SearchBST
```

PSN. Inserting into a BST

Write a recursive function InsertBST for inserting into a BST.

```
Public void InsertBST(Node data, Node search){
    It is assumend you declared all the data before hand into a new nod
    If(search == NULLPTR){
        //insert node here
        Search = data;
    }else{
        If(search->key > data->key){
            insertBST(search->left, data);
        }else if(search->key <= data->key){
            InsertBST(search->right, data);
        }
    }
}

Again very simmilar to what he has
```

Insertion into a BST

```
void InsertBST(ptrNode Root, KeyType NewKey)
{ // inserts NewKey into the binary search tree
    // create a new node for NewKey
    ptrNode NewPtr = new Node;
    NewPtr -> key = NewKey; NewPtr -> Left = NULL; NewPtr -> Right = NULL;
    if (Root == NULL)
        Root = NewPtr;
    else
        if (NewKey < Root -> Key)
            if (Root -> Left == NULL) Root-> Left = NewPtr;
            else InsertBST(Root -> Left);
        else
            if (Root -> Right == NULL) Root-> Right = NewPtr;
            else InsertBST(Root -> Right);
} //END InsertBST
```

Preorder traversal of a BST allows for the binary tree to be rebuilt

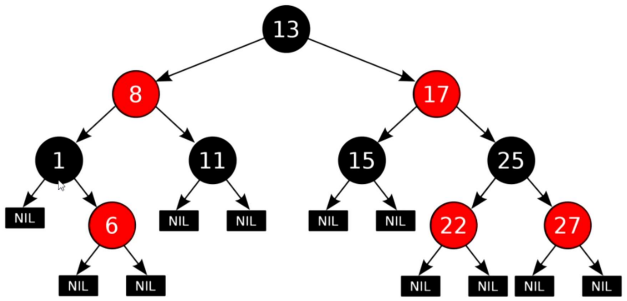
String -> tree (BST) -> String

Deleting from BST

if its a leaf node then just delete
if it has a child then replace the node with a child
if it has 2 children then replace the node with the inorder successor

inorder successor is next node in inorder transversal

Red-black Tree
RB tree is a BST
ether a node can be red or black
the root is black but can be changed
if a node is red both of the children are black
it's decent to any null node (which is black) will encounter the same amount of black nodes every time
All null nodes are black



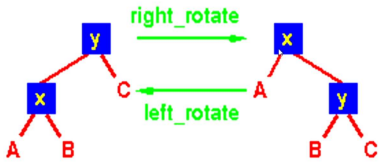
Red-Black tree is balanced

depth = O(log n)

Red-Black tree is balanced bc

insertion works the same but rotation may be needed to preserve BST property

you would then re color the tree up the tree



analysis of insertion
depth O(log n) so search takes O(log n)
rotations take O(1)
So insertion takes O(log n)

Deleting in a RB tree is the same thing for a BST
if the node was red then no rules are violated
if the node is black then recoloring may be needed

in general insertion and deletion is kinda the same for BST but you have to rotate/recolor the tree to keep it balanced

Analysis of deletion
deletion takes O(log n) time

Priority Ques and Heap Sort

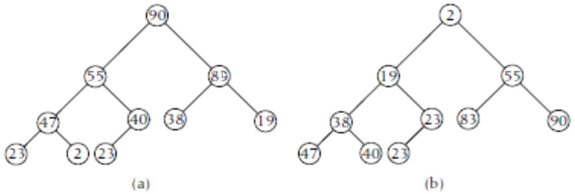
PQ - same operations as a que deque will dequeue with the higest priority

Priority Queue implemented
- List
- Sorted list
- Heap

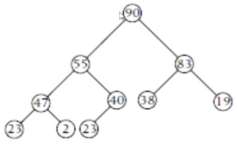
ADT	Enqueue	Dequeue	Change Priority
List	$O(1)$	$O(n)$	$O(1)$
Sorted List	$O(n)$	$O(1)$	$O(n)$
Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$

Max heap - Binary tree where any node is greater than or equal to the keys of its children
min heap - Binary tree where any node is less than or equal to the keys of its children

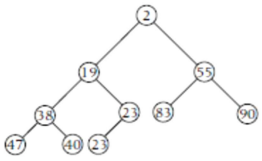
Max-Heap Min-Heap



usely a heap is implemented with an array or linked list like below



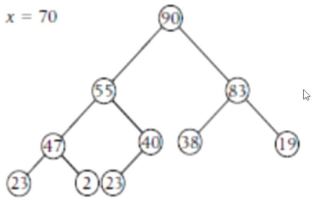
0	1	2	3	4	5	6	7	8	9
90	55	83	47	40	38	19	23	2	23



0	1	2	3	4	5	6	7	8	9
2	19	55	38	23	83	90	47	40	23

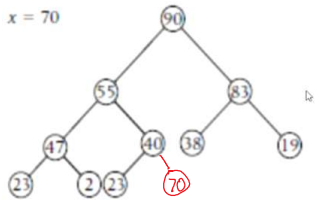
index of parent : $(i-1)/2$
index of child $(2\times i)+1$ and $(2\times i)+2$

PSN. Show action of array for inserting 70 into sample max-heap from previous slide.

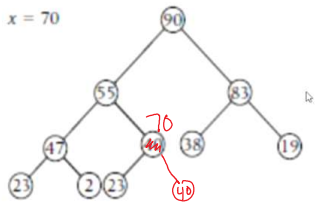


Initial heap

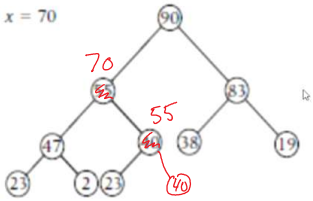
first insert it at the end of the list



70 > 40 so flip



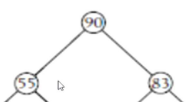
70 > 55

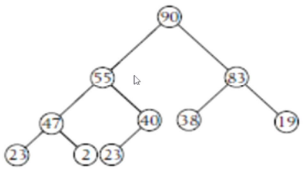


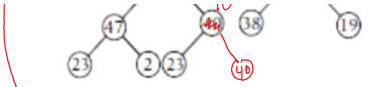
it is now balanced

the list values are relabeled to pic above

PSN. Show action of deleting an element from a the sample max-heap from previous slide.

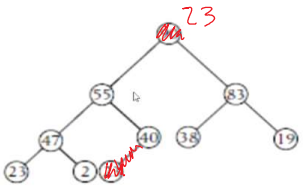




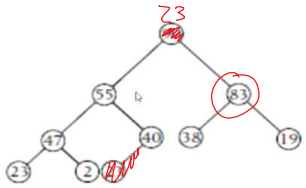


are relevant to pic above

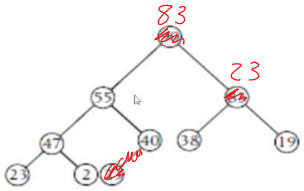
you first start by deleting the head element and bring the last element in to fill in its place



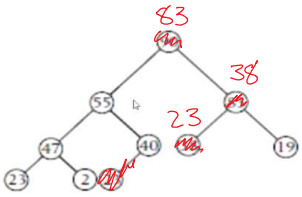
check to see if one child is bigger than the other



Swap the elements of head and biggest child



you then repeat this untill your max heap is correct



all indexes of heap are the same as before

Changing priority - important to know

Complexity analysis - $O(\log_2 n)$

Implementing Sets (Disjoint Sets)

when a collection of sets are disjoint then the intersection of any pair of set is empty
So we only need to implement union and find

implementation of find

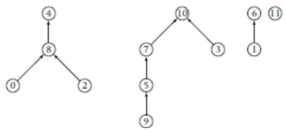
```
procedure Find1(Parent[0:n − 1],x,r)
Input: Parent[0:n − 1] (array representing disjoint subsets of S)
       x (an element of S)
Output: r (the root of the tree corresponding to the subset containing x)
       r ← x
       while Parent[r] ≥ 0 do
            r ← Parent[r]
       endwhile
end Find1
```

PSN. Action of Find

For the sample forest below

a) Show action of Find(9).

b) Show action of Find(2).



2/16

Monday, February 21, 2022 11:10 AM

2/18

Monday, February 21, 2022 11:10 AM