Basics

$b^y = x$ is $\log_b x = y$

Nodes in BST ≤ 2^h-1

Number of leaves = n! for decision tree

Proof By Induction

1. Base Case
2. Since it holds for one n, it could hold for k
3. Prove it holds for k+1

Sorting Algorithms

Insertion Sort: Start with A[i], and compare to A[0] to A[i-1]. Place in proper spot. Then do the same thing for A[i+]

Mergesort: Trivial to divide, but not to merge.

$$T(n) = \begin{cases} \Theta(1) \text{ if n < 1} \\ T\left(\left\lfloor n/2 \right\rfloor\right) + T\left(\left\lceil n/2 \right\rceil\right) + \Theta(n) \text{ if n} \geq 2 \end{cases}$$

Quicksort: Trivial to merge, but not to divide.

	Ω	Θ
Quicksort	nlogn	n^2
Bubblesort	nlogn	n^2
Insertionsort	nlogn	n^2
Heapsort	nlogn	nlogn
Shellsort	nlogn	$n^{1.5}$

	Binary Heap	Binomial Heap	Fibonacci Heap
insert	θ(logn)	O(logn)	θ(1)
min	θ(1)	O(logn)	θ(1)
extractmin	θ(logn)	O(logn)	θ(logn) am
dcrskey	θ(logn)	O(logn)	θ(1)
delete	θ(logn)	O(logn)	θ(logn) am
union	θ(n)	O(logn)	θ(1)

Asymptotic Bound

let $f \bullet g : \mathbb{N} \to \mathbb{R}^+$, and $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists, then

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then $f \in o(g)$

2. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ then $f \in \omega(g)$

3. $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}$ then $f \in \Theta(g)$

Guess and Test

$$T(n) = \begin{cases} 2T\left(\left\lfloor n/2 \right\rfloor\right) + dn \text{ n} \geq 2 \\ 1 \text{ n=1} \end{cases}$$ Guess: $T(n) \leq cn\log_2 n$

Test: Pick unspecified n

Assume for k = …,n-1

$$T(k) \leq ck\log k$$

$$\begin{aligned} T(n) &= 2T\left(\left\lfloor n/2 \right\rfloor\right) + dn \\ &\leq 2\left(c\left\lfloor n/2 \right\rfloor \log\left\lfloor n/2 \right\rfloor\right) + dn \\ &= cn(\lg n - 1) + dn \\ &= cn(\lg n + (d-c)n) \\ &= cn\lg n \end{aligned}$$

so long as c ≥ d, [(d-c)n]

Finding the base case: n=1 won't work

n=2: $T(2) = 2T(1) + d2 = 2d + 2$

$$c2\lg 2 = 2c$$

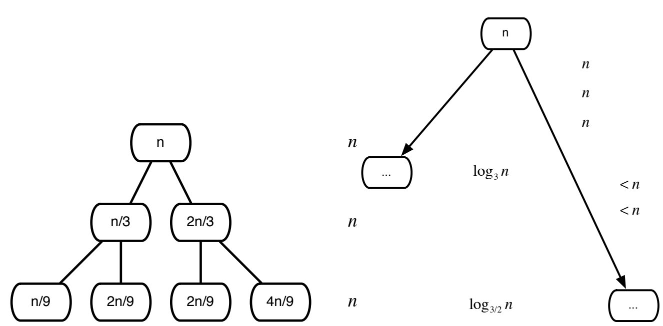
so $T(2) \leq c2\lg 2$ as long as c ≥ d + 1

So $T(n) \leq cn\lg n$ for n ≥ 2 as long as c = d + 1

Recursion Tree

1. Draw a tree representing the recursion
2. Determine how much work each instance of algorithm does
3. Add up work done one each level of the tree
4. Add up work done on all levels

$$T(n)\begin{cases} T\left(n/3\right) + T\left(2n/3\right) \\ 1 \text{ if n} \leq 2 \end{cases}$$



Upper bound: $T(n) \leq n\log_{3/2} n = \frac{1}{\log_3 1.5} n\log n$

Lower bound: $T(n) \geq n\log_3 n$ So $T(n) \in \Theta(n\log n)$

Master Method

$$T(n)=\begin{cases} aT(n/b)=f(n) \text{ if } n\ge n_0 \\ \Theta(1) \text{ if n} < n_0 \end{cases}$$

1. If $f(n)\in O\left(n^{(\log_b a)-\varepsilon}\right)$ for some $\varepsilon>0$ then $T(n)\in\Theta\left(n^{\log_b a}\right)$
2. If $f(n)\in\Theta\left(n^{\log_b a}\log^k n\right)$ for some constant $k\ge 0$, then $T(n)\in\Theta\left(n^{\log_b a}\log^{k+1} n\right)$
3. If $f(n)\in\Omega\left(n^{(\log_b a)+\varepsilon}\right)$ for some $\varepsilon>0$ and $af\left(\frac{n}{b}\right)<\delta f(n)$ for some $0<\delta<1$ and all n large enough. then $T(n)\in\Theta(f(n))$

Amortized Analysis

$$\Phi(D_i)=\Phi(D_{i-1})+COSTam(OP_i)-COSTreal(OP_i)$$

$$\underbrace{\sum_{i=1}^{n} COSTreal(OP_i)}_{\text{calculating running time is hard}}\le\underbrace{\sum_{i=1}^{n} COSTam(OP_i)}_{\text{calculating running time is easy}}$$

Binomial Heaps

Properties: binomial tree of order k nodes

1. height of k
2. $\binom{k}{i}$ nodes at depth i

Fibonacci Heaps

Analysis: $\Phi(H_i)=\underbrace{u}_{\text{constant}}\;[\underbrace{t}_{\text{\# of tees}}(H_i)+2\underbrace{m}_{\text{\# of marked nodes}}(H_i)]$

Dynamic Programming

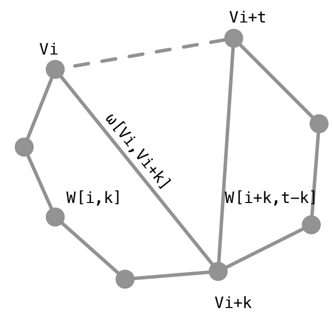
- Given set $S=\{s_1,\ldots,s_n\}$, where s has weight and length
  - we want set $C\in S$ that maximizes $\sum w(s)$
- **Step 1:** Determine subproblem to solve, ie look at s_1,s_2,s_3 Then write recurrence relation for solution to the problem as a function of solutions to the the subproblem.
  - Assume $f(i)\le\ldots\le f(n)$ then

    if i_n is not chosen $W[n]=W[n-1]$

    if i_n is chosen, $w(i_n)+W[p(n)]$

    where p(n)= interval number that ends before i_n starts.
  - Store solutions to the subproblem in a BST
- **Step 2:** Design a table and decide how you'll store the subproblems.
  - eg: array W[3] has solution when picking s_3
- **Step 3:** Write the code.
  - Multiple loops for multiple dimensions

Minimum Weight Convex Polygon



$$\min_{1\le k\le t-1}\left\{\omega(V_i,V_{i+k})+\omega(V_{i+k},V_{i+t})+W[i,k]+W[i+k,t-k]\right\}$$

NP Complete

- **Cook's Theorem**: if SAT can be solved in $O(n^k),k\le 0$ then all problems in NP can be too.

  And if all problems in NP can be solved, it's called NP-complete.
  - This is hard to prove, so we use simpler method
    - *If a problem is in NP, and you can solve the problem in poly time, then that problem is in NP-complete*
- How to prove that problem P is in NP complete
  - 1. Show that P is in NP
    - Show that given an answer, you can can verify it in $O(n^k),k\le 0$.
  - 2. Prove that if P $\in O(n^k),k\le 0$ then all problems in NP can be solved in $O(n^k),k\le 0$
    - **Polynomial-time reduction:** used when we know a problem is in NP-complete
      - Pick known problem, P_{NPC}
      - Give algorithm that transforms instances of P_{NPC} to P with a yes or no answer
        - Solve P and give the same answer answer as P_{NPC} ∎