Algorithms, Good Algorithms

Thursday, September 22, 2022 3:26 PM

Algorithms:

- Procedure for performing computation
- Broken into steps
- Inputs/Outputs that are finitely describable

Good Algorithms:

- Must produce correct answer
- In reasonable time
- In reasonable space
- With less energy
- Etc

CSE 101: Focus on time efficiency

Modification vs Reduction

Wednesday, September 28, 2022 4:13 PM

Modification: modify existing algorithm to solve the new problem

Reduction: reduce the input space such that an unmodified existing algorithm can solve the new problem

Example:
Given a graph where each node is labeled {0, 1} and s,t in V. Find an alternating path from s to t.

Modification:
Modify DFS such that a vertex is recursively called only if it is different from the current vertex
Proof: need to prove both
- Any node v visited has a path from s to v that alternates
- Any node v not visited does not have a path from s to v that alternates

Reduction:
Removing edges from vertex with labels (0, 0) and (1, 1)
Proof: need to prove both
- If there is a path from s to t with alternating labels, then the algorithm returns true
- If there is not a path from s to t with alternating labels, then the algorithm returns false

Prefer reduction over modification

How to perform reduction:
1) Modify the input
2) Run solution to another problem
3) Check output of step 2 and decide correct return value

Runtime Notations

Thursday, September 22, 2022 4:35 PM

X in O(Y) : X <= Y (upper bound)

X in Ω(Y) : X >= Y (lower bound)

X in Θ(Y) : X == Y (tight bound)

X in o(Y) : X < Y

X in w(Y) : X > Y

Store & Re-use (Dynamic Programming)

Thursday, September 22, 2022 4:17 PM

If the algorithm is recomputing values, store and re-use values
Basis for dynamic programming

Graphs, Undirected Connectivity, DFS

Tuesday, September 27, 2022 3:25 PM

Terminology: G = (V,E) where
V: set of vertices/nodes
E: set of edges which are pairs of vertices

Directed Graphs: E are ordered pairs
Undirected Graphs: E are unordered pairs

Tree Edge: edge traversed by DFS
Back Edge: edge not traversed by DFS

Graph storage methods:

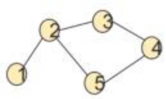
Adjacency matrix

V x V matrix A

A(i,j) = 1 if (i,j) is in E
0 otherwise

Symmetric if G undirected

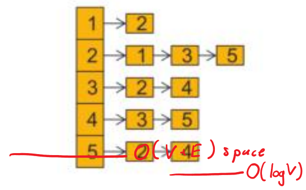
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$



PRO check for an edge in O(1) time
CON uses up $O(V^2)$ space

Adjacency list

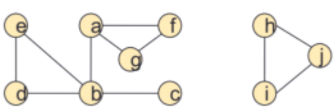
For each node, list of outgoing edges



PRO just O(E) space
CON check for an edge in O(V) time
PRO easily iterate through node's neighbors

Connected Graphs:

An undirected graph is *connected* if there is a path between any pair of nodes.



This graph has 2 *connected components*.

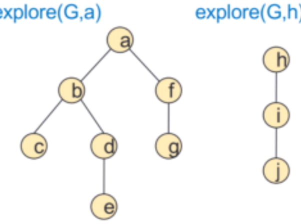
explore(G,v) returns the connected component containing v.

To explore the rest of the graph, restart explore() elsewhere.

DFS decomposes an undirected graph into its connected components!

```
procedure dfs(G)
for all v in V:
  visited[v] = false
for all v in V:
  if not visited[v]:
        explore(G,v)
```

explore(G,a) explore(G,h)

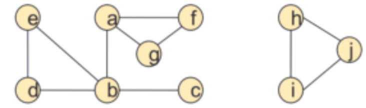


Graph Explore: Find all nodes accessible from v

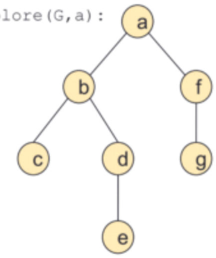
```
procedure explore(G,v)

input: graph G = (V,E); node v in V
output: visited[u] is set to true
  for all u reachable from v

visited[v] = true
previsit(v)
for each edge (v,u) in E:
  if not visited[u]:
        explore(G,u)
postvisit(v)
```



explore(G,a):



Depth-First Search: Decompose graph into connected component

```
procedure dfs(G)
for all v in V:
   visited[v] = false
for all v in V:
   if not visited[v]:
         explore(G,v)
```

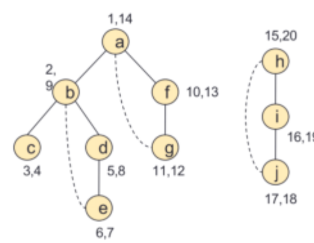
Runtime: O(V+E)
each vertex is visited once during the outer loop

each edge is traversed twice during the inner loop

Modifying using previsit and postvisit:

```
procedure previsit(v)
pre[v] = clock++
procedure postvisit(v)
post[v] = clock++
```

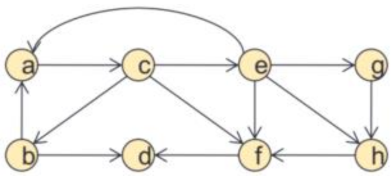
pre[v] = initial time of discovery
Post[v] = time of final departure

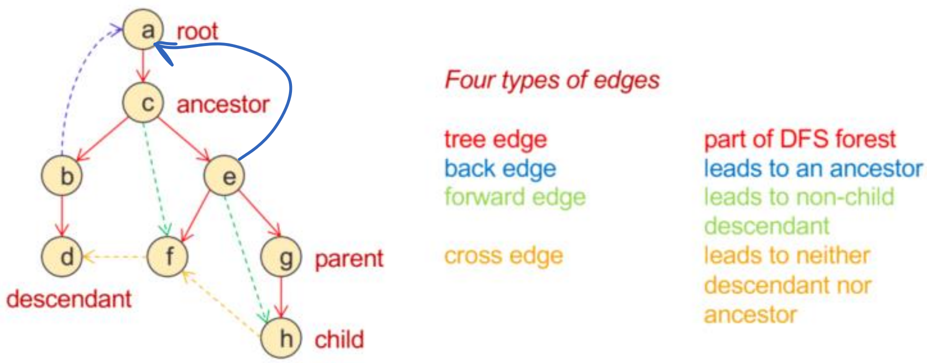


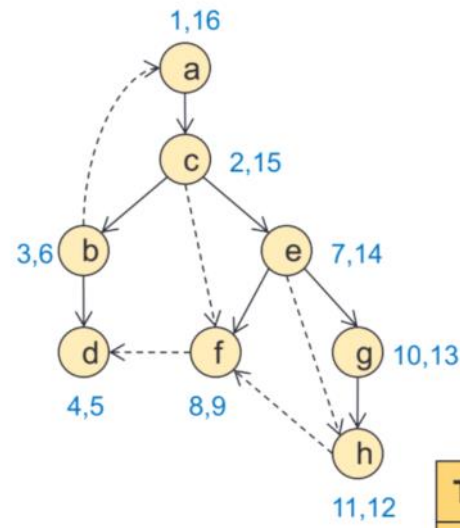
Directed DFS & Terminology

Thursday, September 29, 2022   3:32 PM

Directed DFS: Basically the same as DFS, but edge direction matters







Note: Where the root node is the starting node

Ancestor - Descendent: There is a path from the ancestor to descendent
Parent - Child: Ancestor descendent pair that are one edge apart

Def: Pre/Post Signature of Ancestors

Type of edge	pre/post criterion for edge (u,v)
Tree	pre[u] < pre[v] < post[v] < post[u]
Forward	pre[u] < pre[v] < post[v] < post[u]
Back	pre[v] < pre[u] < post[u] < post[v]
Cross	pre[v] < post[v] < pre[u] < post[u]

Note: undirected DFS can only have Tree, Forward/Back edges

Cycles

Thursday, September 29, 2022 3:56 PM

Def: A <u>cycle</u> is a circular path in a directed graph

Def: A graph is <u>acyclic</u> iff it has no cycles

Proof: A directed graph G has a cycle iff DFF encounters a back edge

(() Suppose DFS encounters a back edge from node v to node u.

Then G has a cycle consisting of the path from u to v in the search tree, plus edge (v,u).

()) Suppose G has a cycle

$$v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k \rightarrow v_0$$

Let v_i be the first of these nodes to be explored; then the rest of them lie in the DFS subtree below v_i; and (v_{i-1}, v_i) (or (v_k, v_0) if i=0) is a back edge.

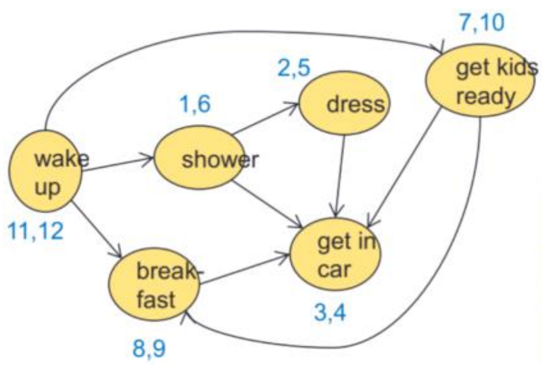
DAGs, Topological Ordering, Source & Sink

Thursday, September 29, 2022 4:05 PM

Def: A <u>DAG</u> or Directed Acyclic Graph

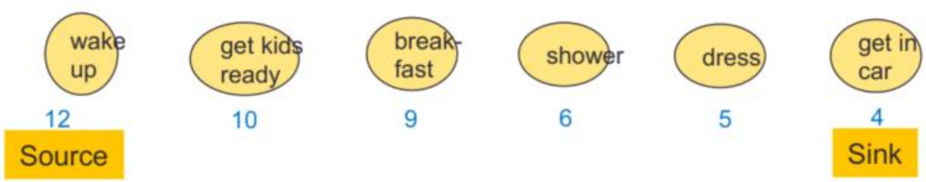
Idea: Topological ordering
- We can use a DAG to find the order of causal things
  Ex: In what order should tasks be performed

Compute POST numbers



<u>Claim</u> In a DAG, every edge leads to a lower post number.
<u>Proof:</u> The only edges (u,v) for which post[v] > post[u] are back edges.
And a DAG has no back edges!

Type of edge	pre/post criterion for edge (u,v)
Tree	pre[u] < pre[v] < post[v] < post[u]
Forward	pre[u] < pre[v] < post[v] < post[u]
Back	pre[v] < pre[u] < post[u] < post[v]
Cross	pre[v] < post[v] < pre[u] < post[u]



Arrange in descending order of POST numbers

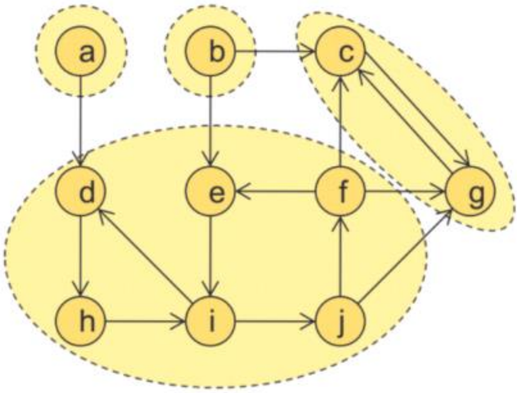
Def: A <u>Source</u> is a node with <u>no incoming edges</u>. A <u>Sink</u> is a node with <u>no outgoing edges</u>.

SCCs, Directed Connectivity

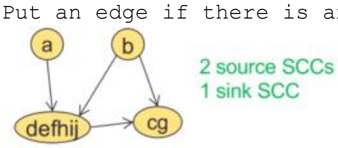
Thursday, September 29, 2022 4:16 PM

Def: In directed graphs, u and v are <u>connected</u> iff there is a path u -> v and a path v -> u

Def: Strongly Connected Components are subgraphs where all nodes mutually connected



The <u>Metagraph</u> is the DAG of SCCs
- Shrink each SCC into meta-nodes
- Put an edge if there is any edge between two nodes in two meta-nodes



Every directed graph is the DAG of its strongly connected components.

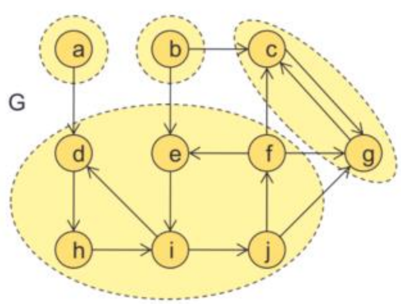
Two-tiered structure of directed graph:
Top level: DAG, very simple structure
Finer detail: peek inside one of the meta-nodes

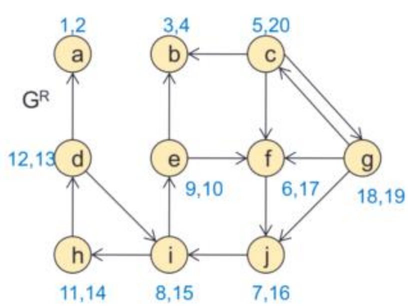
Finding SCCs:
- Property 1: If we run explore on a <u>Sink</u> SCC, then we will precisely identify the SCC
- Property 2: The node with the highest <u>post</u> number, it will be a <u>Source</u>
- Property 3: If components C and C' such that there is an edge from C to C'
  then the highest <u>post</u> in C > highest <u>post</u> in C'

Algorithm: Finding SCCs

Create a new graph G^R by reversing all edges in G

```
run DFS on G^R
for v in V, in decreasing order of G^R-post numbers:
   if not visited[v]:
         explore(G,v)
         output nodes seen as a SCC
```





Ordering from G^R: c,g,f,j,i,h,d,e,b,a

Note: SCC(G) == SCC(G^R)

Shortest Paths, BFS, Dijkstra's

Tuesday, October 4, 2022 3:33 PM

Idea: any node of distance d+1 must come from node of distance d

Def: Breadth First Search

```
procedure bfs(G,s)

input: graph G = (V,E); node s in V
output: for each node u, dist[u] is
   set to its distance from s

for u in V:
   dist[u] = ∞
dist[s] = 0
Q = [s]  // queue containing just s

while Q is not empty:
   u = eject(Q)
   for each edge (u,v) in E:
       if dist[v] = ∞ :
             inject(Q,v)
             dist[v] = dist[u]+1
```

Time complexity: O(V + E)

Idea: not all edges may have the same weight, use a Priority Queue to compute least cost path

Def: Dijkstra's Algorithm

```
procedure dijkstra(G,l,s)

input: graph G = (V,E); node s;
  positive edge lengths l_e
output: for each node u, dist[u] is
  set to its distance from s

for u in V:
  dist[u] = ∞
dist[s] = 0
H = makequeue(V)  // key = dist[]

while H is not empty:
  u = deletemin(H)
  for each edge (u,v) in E:
    if dist[v] > dist[u] + l(u,v):
       dist[v] = dist[u] + l(u,v)
       decreasekey(H,v)
```

Where decreasekey(H,v) updates the key for v to the best dist[v] seen so far.

Time complexity: O(V + E + V*deletemin + V*insert + E*decreasekey)

Depends on Priority Queue implementation!

NOTE: Only works for positive weights

Priority Queues

Tuesday, October 4, 2022 4:26 PM

When to use:

	Array	Heap
Sparse Graph $E = \Theta(V)$	No $O(V^2)$	Yes $O(V*\log(V))$
Dense Graph $E = \Theta(V^2)$	Yes $O(V^2)$	No $O(V^2*\log(V))$

Def: Array as PQ

- Array(hash table): indexed by vertex, giving key value.

Example: (A,2),(B,9),(C,4),(D,1),(E,6),(F,3),(G,4)

H[A] = 2, H[B] = 9 ,.....

- deletemin: O(|V|)
- decreasekey: O(1)

Total Dijkstra's runtime: $O(V + E + V*V + E) = O(V^2)$

Def: Binary Heap as PQ
Binary Tree such that each node's children have a less priority key value than itself
Keep supplemental array indexed by V pointing to its position in the Binary Tree

- deletemin: $O(\log(|V|))$
- decreasekey: $O(\log(|V|))$

Total Dijkstra's runtime: $O(V + E + V*\log(V) + E*\log(V)) = O((V + E)*\log(V))$

Def: Fibonacci Heap as PQ
Total Dijkstra's runtime: $O(V*\log(V)+E)$

Bellman-Ford Algorithm (Negative Dijkstras)

Thursday, October 6, 2022 4:25 PM

Idea: We want to use negative edge weights, rather than just update edges connected to the current node, update all nodes with the best distance seen so far

Def: Bellman-Form Algorithm

```
procedure shortest-paths(G,l,s)

input: graph G = (V,E); node s;
  edge lengths l_e
output: for each node u, dist[u]
  is set to its distance from s

for all u in V: dist[u] = ∞
dist[s] = 0

repeat |V|-1 times:
  for all e in E:
     update(e)
```

```
procedure update(edge (u,v))
if dist[v] > dist[u] + l(u,v):
   dist[v] = dist[u] + l(u,v)
```

Time Complexity: O(|V| * |E|)

Def: Better Shortest Path (use topological sort)

```
procedure
  dag-shortest-paths(G,l,s)
input: dag G = (V,E); node s; edge
  lengths l_e
output: for each node u, dist[u]
  is set to its distance from s

for all u in V: dist[u] = 1
dist[s] = 0

topologically sort G
for nodes u in topological order:
  for all (u,v) in E:
     update(u,v)
```

```
procedure update(edge (u,v))
if dist[v] > dist[u] + l(u,v):
   dist[v] = dist[u] + l(u,v)
```

Minimum Spanning Trees

Tuesday, October 11, 2022 3:26 PM

Problem: We want to create a tree from a connected undirected graph such that the sum of edge weights is minimal. That is, we want to find the minimum edges to connect all nodes in a graph.

Prim's Algorithm: pick the lightest edge that keeps the graph connected and does not create a cycle

```
procedure Prims(G,l,s)

input: graph G = (V,E); node s;
  edge lengths l_e
output: MST

for u in V:
  cost[u] =∞
cost[s] = 0
H = makequeue(V)  // key = cost[]

while H is not empty:
  u = deletemin(H)
  for each edge (u,v) in E:
    if cost[v] > l(u,v):
        cost[v] = l(u,v)
        decreasekey(H,v)
```

Runtime: Basically djikstra's but G must be connected $E=\Omega(V)$

Binary Heap: $O(E*\log(V))$

Array: $O(V^2)$

Kruskal's Algorithm: pick the lightest edge that doesn't create a cycle

```
Procedure Kruskal(G,w):

   for all v in V:
      makeset(v) // add each verex in its own set

   X = {}
   sort E in increasing order by weight
   for edges (u,v) until |X| = |V| - 1:
      if find(u) != find(v):
         add edge (u,v) to X
         union(u,v)
```

Cut Property

Tuesday, October 11, 2022 4:13 PM

Any algorithm which creates an MST must fulfil the cut property:

Claim: Let $X \subset E$ be part of some MST T of G=(V,E).

Pick a subset of nodes $S \subset V$ such that T has no edges between S and V-S.
Let e be the lightest edge between S and V-S.

Then $X \cup \{e\}$ is part of an MST, T'

Idea: Given subsets of vertices S and V-S, then the lightest edge connecting the two subsets is part of an MST

Disjoint Set Data Structures

Thursday, October 13, 2022 3:29 PM

Def: A Disjoin Set has the following operations:

makeset(S): put each element in S into a set by itself
find(u): returns which set contains u
union(u,v): unions the two sets containing u and v

Implementations:

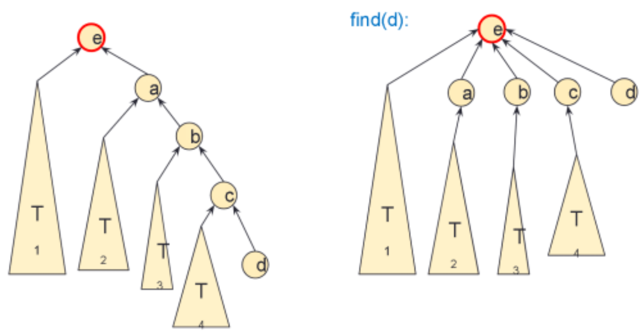
Tree:
Keep a tree where a node represents a tree, and all children are part of that set. Each node will have a parent and rank.

For makeset(V): set all parent pointers to nil and rank to 0
For find(u): iterate through parents to find the top most node
For union(u,v): set the least rank root node parent to the most rank root node
Update ranks as needed

Runtime:
makeset(S): O(S)
find(u): O(log(V))
union(u,v): O(log(V))

Kruskal: O((V+E)*log(V))

Path Compression: Using a tree, we can set the parent of all nodes encountered in find(u) directly to the root node:




```
procedure makeset(x)
p[x] = x
rank[x] = 0

procedure find(x)
while x ≠ p[x]:
   x = p[x]
return x

procedure union(x,y)
rootx = find(x)
rooty = find(y)
if rootx = rooty: return
if rank[rootx] > rank[rooty]:
   p[rooty] = rootx
else:
   p[rootx] = rooty
   if rank[rootx] = rank[rooty]:
      rank[rooty]++
```

Optimization, Global/Local Search, Greedy Algorithms

Thursday, October 20, 2022 3:29 PM

Optimization problems:

- Find the nest solution from a large space of possibilities
- May have constraints on solution
- Must have an objective way to judge solutions

Global Search / Exhaustive: search all possible solutions to find the best

Local Search: Break the global search into series of simpler local search

Greedy Algorithms: Reach the optimal solution by taking the optimal decision every time

Proving Correctness:
Let I be any instance of our problem, GS be the greedy algorithm's solution, and OS be any other solution.
If minimization: show Cost (OS) $\geq$ Cost(GS)
If maximization: show Value(GS) $\geq$ Value (OS)

Bipartite Matching

Thursday, October 27, 2022 3:59 PM

Bipartite: Graph such that there is a set S and all edges go from S to V - S

Matching: Given a bipartite graph, select a set of edges such that each node has degree 1

Divide and Conquer

Tuesday, November 1, 2022 3:36 PM

Idea: Break problem into smaller subproblems and recursively solve

Exchange Argument

Tuesday, October 18, 2022 4:34 PM

1. Let G be a greedy solution and g be a greedy choice the algorithm makes

2. Let OS be a solution achieved by not choosing g

3. Show how to transform OS into OS' that chooses g and is at least as good as OS
    - Must show OS' is valid
    - Must show OS' is better than OS

4. Use 1-3 to move closer to G OR Use 1-3 in induction to show that we can always make choices consistent with G

Greedy Stays Ahead

Thursday, October 27, 2022 4:39 PM

Define a progress measure

Show that the greedy solution is ahead in the progress measure compared to any arbitrary solution at all points

Use to establish the optimality of the algorithm

Master Theorem

Thursday, November 3, 2022 3:57 PM

Master Theorem: If $T(n) = aT(n/b) + O(n^d)$ for some constants $a > 0, b > 1, d \geq 0$,

Then

$$T(n) \epsilon \begin{cases} O(n^d) & if \ \ a < b^d \\ O(n^d \log n) & if \ \ a = b^d \\ O(n^{\log_b a}) & if \ \ a > b^d \end{cases}$$

Top-heavy ($a < b^d$)

Steady state ($a = b^d$)

Bottom heavy ($a > b^d$)

MergeSort

Tuesday, November 1, 2022 3:38 PM

Idea: Take two sorted arrays and combine them into larger sorted array

Merge(A[1..n], B[1..n]): linear time, combines two sorted lists

- I:=1 ; J:=1
- FOR k=1 TO 2n do:
- IF I > n THEN C[k]:= B[J]; J++
- ELSE IF J > n THEN C[k]:= A[I];I++
- ELSE IF A[I] > B[J] THEN C[k]:=B[J]; J++
- ELSE C[k]:=A[I];I++
- Return C

MergeSort (A[1..n])

- IF n=1 return A[1]
- ELSE Return Merge (MergeSort(A[1..n/2]),MergeSort (A[n/2+1…n])

Time analysis:
Merge: O(n)
MergeSort: T(n) = 2T(n/2) + n
O(n*log(n))

Fast Multiply

Tuesday, November 1, 2022 3:47 PM

Idea: Perform partial products and then combine, we will also leverage the fact that addition is cheaper than multiplication

function **multiplyKS** (x,y)
Input: n-bit integers x and y
Output: the product xy

- If n=1: return xy
- x_L, x_R and y_L, y_R are the left- and right-most n/2 bits of x and y, respectively.
- $R_1 = \textbf{multiplyKS}(x_L, y_L)$
- $R_2 = \textbf{multiplyKS}(x_R, y_R)$
- $R_3 = \textbf{multiplyKS}\big((x_L+x_R)(y_L + y_R)\big)$
- return $R_1 * 2^n + (R_3 - R_1 - R_2) * 2^{\frac{n}{2}} + R_2$

Runtime:
T(k) = 3T(k/2) + O(k)
Max k = log(n)
Total: O(3^log(n)) = O(n^log(3))

Selection, Quicksort, QuickSelect, MedianOfMedians

Thursday, November 3, 2022 3:54 PM

Select

Problem: Given a list of numbers, find the kth largest element

Idea: We can pick a random pivot and separate into groups of values smaller (SL), equal (Sv), and larger (SR) than the pivot
If k ≤ |SL| then k in SL
If k ≤ |SL| + |Sv| then k in Sv
If k > |SL| + |Sv| then k in SR

- Input: list of integers and integer k
- Output: the kth smallest number in the set of integers.

- function Selection(a[1…n],k)
- if n==1:
  - return a[1]
- pick a random integer in the list v.
- Split the list into sets SL, Sv, SR.
- if k≤|SL|:
  - return Selection(SL,k)
- if k≤|SL|+|Sv|:
  - return v
- else:
  - return Selection(SR, k-|SL|-|Sv|)

Runtime:
In the best case, |SL| = |SR| then T(n) = T(n/2) + O(n) and runtime is O(n)

In the worst case, v is the minimum then T(n) = T(n-1) + O(n) and runtime is O(n^2)

Quicksort

- procedure quicksort(a[1…n])
- if n≤1:
  - return a
- set v to be a random element in a.
- partition a into SL,Sv,SR
- return quicksort(SL)∘Sv∘ quicksort(SR)

Runtime: Since we need to recurse on both sides, the runtime can be approximated to O(nlogn)

QuickSelect

Idea: split array into sets of 5 and find medians of sets. Then find medians of medians by recursion.

- MofM(L,k)
- If L has 10 or fewer elements:
  - Sort(L) and return the kth element
- Partition L into sublists S[i] of five elements each
- For $i = 1, \ldots n/5$
  - $m[i]$ =MofM(S[i],3)
- M = MofM($[m[1], \ldots, m[n/5]]$,$n/10$)

Runtime: T(n) = T(n/5) + T(7n/10) + O(n) -> O(n)

Backtracking, Maximum Independent Set

Thursday, November 10, 2022 3:32 PM

Scope: problems asking to find the optimal solution in a large solution space

Idea: Like D&C, we can solve a smaller subproblem. But, Backtracking usually reduces problem by constant size rather than factor

Ex: Maximal independent set
Given graph G with, find the largest set such that no two members are connected by an edge

Solution: On some decision to pick vertex V then:
- If we pick V, then recurse on G - {A ∪ A's neighbors}
- If we don't pick V, then recurse on G - {V}
- Additionally, if degree(V) = 0 or 1 then we will always pick V anyways

```
MIS3(G, undirected graph)
  if |V| = 0:
     return Ø
  pick a vertex v.
  In = MIS3(G – {v and all of v's neighbors}) ∪ {v}
  if deg(v) = 0 or deg(v) = 1:
    return In
  Out = MIS2(G – {v})
  If |In|>|Out|:
    return In
  else:
    return Out
```

Runtime: T(n) = T(n - 1) + T(n - 3) + O(n) : O(1.46^n)

Weighted Event Scheduling

Thursday, November 10, 2022 4:35 PM

Ex: Given the event scheduling problem, add weights to each event and try to maximize the total weight of the schedule

Solution: Sort the events by end time. Pick the last ending event and recurse on the two cases
- If the event is included, recurse on the schedule without all conflicting events
- If the event is not included, recurse on the schedule without this event

BTWES($I_1, \ldots, I_n$) (sorted by end times.)
  if $n = 0$: **return** 0
  if $n = 1$: **return** $value(I_1)$
  OUT = **BTWES**($I_1, \ldots, I_{n-1}$) T(n-1)
  Let I_k be the last event to end before I_n starts.
  IN = **BTWES**($I_1, \ldots, I_k$) $+ value(I_n)$ T(k)
  return max(OUT,IN)

Runtime: T(n) = 2T(n-1) + O(n) = O(2^n)

Note: All recursive calls are the form (I1 … Ik), so there are only n-1 total calls

Memoization

Thursday, November 17, 2022 3:54 PM

When performing backtracking, save all intermediate steps so repeated steps are not recomputed

Ex: for Weighted Event Scheduling: create array and store intermediate steps (I1 … Ik) at index k

Dynamic Programming

Thursday, November 17, 2022 3:57 PM

1) Define subproblems are corresponding array
2) Define bases cases
3) Define recursion for sub problems (case analysis)
4) Order the subproblems
5) Define final output
6) Put all together in iterative algorithm that fills in the array

Ex: Find the max value among all valid schedules of (I1 … In)
1) Let A[k] be the max value among all valid schedules of (I1 … Ik)
2) A[0] = 0
3) Case 1: Ik is in the max schedule, A[k] = value(Ik) + A[j] where j is the last interval to end before Ik starts
   Case 2: Ik is not in the schedule, A[k] = A[k-1]
   A[k] = max(Case 1, Case 2)
4) Since each subproblem is dependent on smaller index, order 0 to n
5) Final output = A[n]

6)

MaxSubset($I_1, \dots, I_n; v(I_1), \dots, v(I_n)$) ordered by end times.

$A[0] = 0$
for $k = 1 \dots n$:
  $j = 1$
  while End(I_j) $\leq$ Start(I_k):
    $j = j + 1$
  $A[k] = \max(A[k-1], v(I_k) + A[j-1])$
return $A[n]$

Ex: Given items with value v[1] … v[n] and weight w[1] … w[n] and max weight of C
1) Let A[j, b] be the max value given items 1 … j with max weight b
2) A[j, 0] = 0, A[0, b] = 0
3) Case 1: Item j is in the max for weight b, A[j, b] = v[j] + A[j, b - w[j]]
   Case 2: Item j is not in the max weight b, A[k, w] = A[j - 1, b]

The cell $[j, b]$ is dependent on $[j, b - w[j]]$ and $[j - 1, b]$

							$(j-1, b)$
		$(j, b - w[j])$					(j, b)

4) So, you can order the problems by filling in each row from left to right starting from the top row and going down.

FOR $j = 1 \dots n$
- FOR b= $1 \dots C$

5) Final output = A[n, C]

6)

Knapsack($w[1 \dots n], v[1 \dots n], C$)
$KS[j, 0] = 0$ for all j
$KS[0, b] = 0$ for all b
for j from 1 to n:
  for b from 1 to C:
    if $w[j] > b$:
      $KS[j, b] = KS[j-1, b]$
    else:
      IN = $v[j] + KS[j, b - w[j]]$
      OUT = $KS[j-1, b]$
      $KS[j, b] =$ max(IN, OUT)
return $K[n, C]$