DSA: TOPICS AND ASSORTED INFO

ALGORITHMS

FIBONACCI NUMBERS – DPV 3

Fibonacci numbers are defined as the sum of the previous two Fibonacci numbers.

```
Function Fib(n)
if n = 0: return 0
create an array f[0…n]
f[0] = 0, f[1] = 1
for i = 2…n:
  f[i] = f[i-1] + f[i-1]
return f[n]
```

Runtime: $O(n^2)$, Correctness: DPV 6

MAX SUB-SEQUENCE SUM

Given a sequence of integers $A_1, \dots A_n$, find the maximum value of $\sum_{k=i}^{j} A_k$. Ex. {-1, 11, -4, 13, -5, -2}: 20

```
Function MaxSubsequence(a[1…n])
maxSum = 0, thisSum = 0
for j in 0:n-1
  thisSum = thisSum + a[j]
  if( thisSum > maxSum )
    maxSum = thisSum
  else if ( thisSum < 0 )
    thisSum = 0;
return maxSum
```

Runime: $O(n)$, Correctness: Any negative subsequence cannot be a prefix of the optimal subsequence. We compute the maximum subsequence ending at position j

GENERATING RANDOM PERMUTATIONS – CLRS 101

Find a perfectly random permutation of $A_1, \dots A_n$ with each permutation having a probability of $\frac{1}{n!}$ of appearing.

```
PermuteBySorting(A[1…n])
n = length[A]
for i in 1 to n
  P[i] = Random(1, n^3)
Sort A, using P as sort keys
Return A
```

Runtime: $O(n \log n)$ due to sorting. Correctness : CLRS 101

BASIC ARITHMETIC – DPV 13-15, 47

ADDITION

$O(n)$ where n is the number of bits of the numbers - Optimal.

MULTIPLICATION

Using a Divide and Conquer approach:

```
Function multiply(x,y)
If n=1: return xy
xl, xr = leftmost ceiling(n/2), rightmost floor(n/2) bits of x
yl, yr = leftmost ceiling(n/2), rightmost floor(n/2) bits of y
p1 = multiply(xl,yl)
p2 = multiply(xr,yr)
p3 = multiply(xl+xr,yl+yr)
return p1*2^n+(p3-p1-p2)*2^(n/2) + p2
```

Runtime: $O(n^{1.59})$, Correctness: DPV 47

DIVISION

```
Function divide(x,y)
If x = 0: return (q,r) = (0,0)
(q,r) = divide(floor(x/2), y)
q = 2*q, r = 2*r
if x is odd: r = r + 1
if r >= y: r = r – y, q = q + 1
return (q,r)
```

Runtime: $O(n^2)$, Correctness/Analysis: DPV 15

PRIMALITY – DPV 27

Given an integer N, decide whether or not it is prime.

```
Function primality(N)
Pick positive integers a1, a2, …, ak < N at random
If ai^(N-1) = 1 (mod N) for all i = 1, 2, …, k
  Return true
Else
  Return false
```

Runtime: $O(N)$, Correctness: DPV 27

MERGESORT – DPV 51

Sort a list of numbers in ascending order

```
Function mergesort(a[1…n])
if n > 1:
  return merge( mergesort(a[1…floor(n/2)]), mergesort(a[floor(n/2) + 1…n]) )
else
  return a

Function merge( x[1…k], y[1…l] )
If k = 0: return y[1…l]
If l = 0: return x[1…k]
If x[1] <= y[1]:
  Return x[1] concatenated with merge(x[2…k],y[1…l])
Else
  Return y[1] concatenated with merge(x[1…k],y[2…l])
```

Runtime: $O(n \log n)$, Correctness: DPV 51

MEDIAN-FINDING – DPV 53

Given a list of numbers, find the median of the numbers

```
function select(list[1..n], k)
for i from 1 to k
  minIndex = i
  minValue = list[i]
  for j from i+1 to n
    if list[j] < minValue
      minIndex = j
      minValue = list[j]
    swap list[i] and list[minIndex]
  return list[k]
```

Runtime: $O(n)$, Correctness: DPV 54, 55

QUICKSORT – CLRS 145

Sort a list of numbers in ascending order

```
Function quicksort(A,p,r)
If( p < r )
  Then q = partition(A,p,r)
    quicksort(A,p,q-1), quicksort(A,q+1,r)
Function partition(A,p,r)
X = A[r], I = p-1
For( j = p to r-1 )
  If( A[j] <= x ) i++, exchange A[i] and A[j]
Exchange A[i+1] and A[r]
Return i+1
```

Runtime: $O(n \log n)$, Correctness: CLRS 146, NOTE: To sort array, call `quicksort(A, 1, length(A))`

MATRIX MULTIPLICATION – DPV 56

Multiply two $n \times n$ matrices optimally. Runtime: $O(n^{2.81})$, Correctness: DPV 57

DEPTH FIRST SEARCH (DFS) – DPV 85

Traverse an entire graph G and produce a search forest which represents which nodes can be reached from which nodes. DFS is useful for searching for the existence of a path between two vertices. Back edges connect a descendent to an ancestor which is not its parent. They represent cycles in the graph. Forward edges lead from a node to a non-child descendent in the DFS tree. Cross edges are any edge which isn't a Tree edge, a back edge, or a forward edge.

```
Function dfs(G)
For all vertices in the vertex set
  Visited(v) = false
For all vertices in the vertex set
  if !visited(v): explore(G,v)

Function explore(G,v)
Visited(v) = true
Previsit(v)
For each edge in the edge (v, u) touching this vertex:
  If !visited(u): explore(G,u)
Postvisit(v)
```

Runtime: $O(|V| + |E|)$, Correctness/Analysis: DPV 85

TOPOLOGICAL SORT/LINEARIZATION – CLRS 550

Given a graph G, return a linear ordering of all its vertices such that if G contains an edge (u,v), then u appears before v in the ordering. Every DAG has a linearization, and every graph can be made into a DAG through the use of strongly connected components.

```
Function topologicalSort(G)
Call DFS(G) to compute finishing times f[v] for each vertex v
As each vertex is finished, insert it onto the front of a linked list
Return the linked list of vertices
```

Runtime: $O(|V| + |E|)$, Correctness/Analysis: CLRS 550

STRONGLY CONNECTED COMPONENTS (SCC) – DPV 91

A strongly connected component of a graph G is a subset of vertices in its vertex set which are strongly connected.

```
Function SCC(G)
Run DFS on G^R (G with all edges reversed)
Processing vertices in inverse ordering of their post numbers from DFS, group
vertices which are fully connected together, then remove them from the graph.
```

Runtime: $O(|V| + |E|)$, Correctness/Analysis: DPV 94

'*' means this topic was presented in semester two

BREDTH-FIRST SEARCH (BFS) – DPV 106, CLRS 531

Given a directed or undirected graph G without edge weights, and a vertex, return all vertices reachable from this vertex, as well as the distances from this vertex to all reachable vertices.

```
Function bfs(G,s)
For all vertices in the vertex set
  Dist(u) = infinity
Dist(s) = 0
Q = [s] (queue containing just s)
While Q is not empty:
  U = eject(Q)
  For all edges (u,v) which are in the edge set
    If dist(v) == infinity
      Inject(Q,v)
      Dist(v) = dist(u) + 1
```

Runtime: $O(|V| + |E|)$ - also depends on implementation of queue, Correctness/Analysis: DPV 106, CLRS 531

DIJKSTRA'S ALGORITHM – DPV 108, CLRS 595

Given a directed graph G with all positive edge weights and a vertex in said graph, return the shortest distances from the given vertex to all other reachable vertices in the graph. NOTE: Dijkstra's algorithm does not work on graphs with negative edge lengths

```
Function dijkstra(G,l,s) (graph, edge lengths, source vertex)
For all vertices, u, in the graph,
  Dist(u) = infinity
  Prev(u) = nil
Dist(s) = 0
H = makequeue(V) using dist values as keys
While H is not empty:
  u = deletemin(H)
  For all edges (u,v) in the edge set
  If dist(v) > dist(u)+l(u,v):
    dist(v) = dist(u)+l(u,v)
    prev(v) = u
    decreasekey(H,v)
```

insert adds a new element to the set
decreasekey accommodates the decrease in the key value of an element
delete-min returns the element with the smallest key, removes it from the set
make-queue builds a priority queue out of the given elements with given keys

Implementation	deletemin	insert/decreasekey	$\|V\| \times$ deletemin $+ (\|V\| + \|E\|) \times$ insert
Array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(\log\|V\|)$	$O(\log\|V\|)$	$O((\|V\| + \|E\|)\log\|V\|)$
d-ary heap	$O\left(\frac{d\log\|V\|}{\log d}\right)$	$O\left(\frac{\log\|V\|}{\log d}\right)$	$O\left((\|V\| * d + \|E\|)\left(\frac{\log\|V\|}{\log d}\right)\right)$
Fibonacci heap	$O(\log\|V\|)$	$O(1)$ (Amortized)	$O(\|V\|\log\|V\| + \|E\|)$

Runtime: $O(|V| + |E|)$ - Depends on heap implementation – See above, Correctness/Analysis: DPV 108, CLRS 595

'*' means this topic was presented in semester two

BELLMAN-FORD ALGORITHM – DPV 118, CLRS 588

Given a graph directed graph G and a set of positive or negative edge lengths, as well as a source vertex, find the shortest paths from the source vertex to all other reachable vertices.

```
Function bellmanFord(G,l,s) (graph, edge lengths, source vertex)
For all vertices, u, in the vertex set,
  Dist(u) = infinity
  Prev(u) = nil
Dist(s) = 0
  Repeat |V| - 1 times:
    For all edges in the edge set
      Update(e)

Function update(e)
Dist(v) = min{dist(v), dist(u) + l(u,v)}
```

NOTE: Can be optimized to linear time for DAGs if vertices are processed in linearized order.

Runtime: $O(|V| * |E|)$, Correctness/Analysis: DPV 118, CLRS 588

HEAPSORT – CLRS 136

Given a list of numbers, sort them in ascending order

```
Function heapsort(A)
Buildmaxheap(A)
For i = length(A) down to 2
  Exchange A[1] and A[i]
  heap-size[A] = heap-size[A]-1
  max-heapify(A,1)
```

Runtime: $O(n \log n)$, Correctness/Analysis: CLRS 136

*CUCKOO HASHING

Cuckoo hashing is a method of dealing with collisions in a hash table by placing the new element in its location in the hash table, and moving the element which is currently there to a new location through the use of a secondary hash function.

```
Function insert(x)
If T[h1(x)] = x or T[h2(x)] = x then return;
Pos = h1(x)
Repeat n times
  If T[pos] = NULL then T[pos] = x, return
  x = T[pos]
  if pos = h1(x) then pos = h2(x) else pos = h1(x)
rehash(), insert(x)
```

Runtime: $O(n)$ (expected), $O\left(\frac{1}{n}\right)$(expected) for rehashing, Correctness/Analysis: See "Cuckoo Hashing for Undergraduates", Rasmus Pagh, 2006

*KRUSKAL'S ALGORITHM – DPV 131

Given an undirected graph with edge weights, Kruskal's algorithm outputs a minimum spanning tree defined by the edge set X. Kruskal's algorithm can also use a method called "path compression" which brings amortized time for `find` and `union` down to $O(1)$. Path compression causes any node looked up by `find` to have its parent become the root node.

There are three properties which are maintained in Kruskal's algorithm

- For any x, rank(x) < rank(parent(x))
- Any root node of rank k has at least 2^k
- If there are n elements overlal, there can be at most $\frac{n}{2^k}$ nodes of rank k.

```
Function kruskal(G,w)
For all vertices in the vertex set
  Makeset(u)
X = {}
Sort the edges E by weight
For all edges {u,v} in the edge set
in increasing order of weight:
  If find(u) != find(v):
    Add edge {u,v} to X
    Union(u,v)

Function makeset(x)
Parent(x) = x
Rank(x) = 0

Function find(x)
While x != parent(x)
  x = parent(x)
Return x
```

```
Function find(x)
(Path Compression)
If x != parent(x)
  parent(x) = find(parent(x))
Return parent(x)

Function union(x,y)
(union by rank)
Rx = find(x)
Ry = find(y)
If Rx = Ry: return
If rank(Rx) > rank(Ry):
  Parent(Ry) = Rx
Else
  Parent(Rx) = Ry
  If rank(Rx) = rank(Ry)
    rank(Ry) = rank(Ry) + 1
```

Runtime: $O(|E|\log|V|)$, $O(n\log^* n)$ with path compression, Correctness/Analysis: DPV 132

*PRIM'S ALGORITHM – DPV 137

Given an undirected graph with edge weights, Prim's algorithm outputs a minimum spanning tree defined by the vertex set V.

```
Function prim(G,w)
For all vertices, u, in the vertex set
  Cost(u) = infinity, Prev(u) = nil
Pick any initial node u0
Cost(u0) = 0
H = makeQueue(V) (Priority queue using cost values as keys
While H is not empty:
  V = deletemin(H)
  For each {v,z} in the edge set,
  If cost(z) > w(v,z): cost(z) = w(v,z), prev(z) = v
```

Runtime:$O(|E|\log|V|)$ (using binary heap), Correctness/Analysis/Hand Worked Problem: DPV 139

*HUFFMAN ENCODING – DPV 138

Huffman encoding is a compression scheme which utilizes a greedy approach in order to generate prefix-free binary encodings for symbols in a message. The final Huffman encoding can be represented as a tree, which has symbols which appear the most toward the top of the tree, and symbols which appear the least toward the bottom. Each level down the tree adds another digit onto the Huffman encoding of a symbol at that height in the tree.

Entropy – a measure of how much randomness an encoding has - of a Huffman encoding is defined as

$$\sum_{i=1}^{n} p_i \log \frac{1}{p_i}, n\ possible\ outcomes, p_i = probability\ of\ the\ i^{th}\ outcome$$

```
Function Huffman(f)
(f is an array of [1…n] frequencies)
Let H be a priority queue of integers, ordered by f
For i = 1 to n: insert(H,i)
For k = n+1 to 2n-1:
  i = deletemin(H), j = deletemin(H)
  create a node numbered k with children i, j
  f[k] = f[i]+f[j]
  insert(H,k)
```

Runtime: $O(n \log n)$, if sorted – $O(n)$, Correctness/Analysis: DPV 138, Entropy: DPV 143

*HORN FORMULAS – DPV 144

Given a set of clauses of two types – described below – determine if there is a satisfying assignment of literals which satisfy all clauses.

Two types of clauses which can appear in a horn formula

Implications whose left-hand side is an AND of any number of positive literals, and whose right hand side is a single positive literal. For example: $(z \wedge w) \rightarrow u$

Pure Negative Clauses consisting of an or of any number of negative literals. For example: $(!u \vee !v \vee !y)$

```
Function horn
Set all variables to false
While there is an implication that is not satisfied:
  Set the right-hand variable of the implication to true
If all pure negative clauses are satisfied:
  Return the assignment
Else
  There is no solution
```

Runtime: $O(number\ of\ implications)$, Analysis: DPV 144

*SET COVER – DPV 145

Given a set of elements B, sets $S_1, \ldots, S_m$ which contain B, output a selection of the S_i elements whose union is B

A Greedy implementation is presented on DPV 145, but is not optimal, due to the fact that Set Cover is NP-Complete.

*EDIT DISTANCE – DPV 159

Given two strings, align them with minimal cost. The cost of an alignment is equal to the sum of all mis-matched characters and the number of "gaps".

```
S-NOWY    -SNOW-Y
SUNN-Y    SUN--NY
Cost: 3   Cost: 5
```

A Dynamic programming approach:

```
For i = 0, 1, 2, …, m
  E(i,0) = i
For j = 1, 2, …, n
  E(0,j) = j
For(i = 1, 2, …, m)
  For(j = 1, 2, …, n)
    E(i,j) = min{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+diff(i,j)}
return E(m,n)
```

Runtime: $O(mn)$, Correctness/Analysis: DPV 163

*KNAPSACK – DPV 164

Given a knapsack which can hold a certain weight and a collection of n items of weight w_i and dollar value v_i, what is the most valuable combination of items that can be fit into the bag?

With item repetition:

```
K(0) = 0
For w = 1 to W
  K(w) = max{K(w-wi)
+vi : wi <= w}
Return K(W)
```

Without item repetition:

```
Initialize all K(0,j) = 0 and all K(w,0) = 0
For j = 1 to n:
  For w = 1 to W:
    If wj > w: K(w,j) = K(w,j-1)
    Else: K(w,j) = max{K(w,j-1), K(w-wj,j-1)+vj}
Return K(W,n)
```

Runtime: $O(NW)$, Correctness/Analysis: DPV 167, 168

*CHAIN MATRIX MULTIPLICATION – DPV 170

Given a series of matrices which are able to be multiplied in a given order, multiply them optimally.

```
For i = 1 to n: C(i,i) = 0
For s = 1 to n-1:  for i = 1 to n-s:
  j = i + s, C(i,j) = min{C(i,k)+C(k+1,j)+m[i-1]*m[k]*m[j] : I <= k < j}
return C(1,n)
```

Runtime: $O(n^3)$, Correctness/Analysis: DPV 170

'*' means this topic was presented in semester two

*ALL PAIRS – SHORTEST PATHS – DPV 172

Find the shortest path between every pair of vertices in the graph. The Floyd-Warshall Algorithm does just this.

```
For i=1 to n:
  For j=1 to n:
    Dist(i,j,0) = infinity
For all (i,j) in the edge set
  Dist(i,j,0) = length(i,j)
For k=1 to n:
  For i=1 to n:
    For j=1 to n:
      Dist(i,j,k) = min{dist(i,k,k-1)+dist(k,j,k-1), dist(i,j,k-1)}
```

Runtime: $O(|V|^3)$, Correctness/Analysis: DPV 172, 173

*TRAVELLING SALESMAN PROBLEM (TSP) – DPV 173

Given a fully connected graph with edge weights and a start vertex, find the shortest round trip which visits every vertex and returns back to the original vertex. The problem is known to be NP-Complete, so a dynamic programming approximation is given.

```
C({1},1) = 0
For s = 2 to n:
  For all subsets S containing {1, 2, …, n} of size s and containing 1:
    C(S,1) = infinity
    For all j in S, j != 1:
      C(S,j) = min{C(S-{j},i)+d[ij] : i in S, i != j}
Return min_j C({1,…,n},j)+d[j1]
```

Runtime: $O(n^2 2^n)$, Correctness/Analysis: DPV 173, 174, 175

*INDEPENDENT SETS – DPV 176

A Subset of nodes in a graph are called an independent set if there are no edges between them. We want to find the size of the largest independent set of the subtree hanging from any specific vertex, r.

An algorithm is presented on DPV 176 which runs in $O(|V| + |E|)$

*MAX FLOW – DPV 199

Given a graph with a source and a sink and many connections in between each with capacities – called a flow network – find the maximum flow that can be pushed through the network from the source to the sink

To solve this problem, we start with zero flow, then we choose an appropriate path from s to t, creating a residual graph in the process. A demonstration can be found on DPV 201, and pseudo-code can be found in the notes for Max Flow.

DATA STRUCTURES

GRAPHS

A graph is a data structure which consists of many a set of vertices and a set of edges which connect vertices. Graph come in two types:

- Directed – These graphs have one-way edges which go from one vertex to another, but not back
- Undirected – These graphs have two-way edges which go both ways from a pair of vertices

Definitions:

- Connected – A graph is connected if from any vertex, there exists a path to reach any other vertex.
- Connected Components – A connected subset of vertices in the graph which has no connections to any other connected components in the graph.
- Strongly Connected Component (SCC) – A connected component of a graph which has connections to the rest of the graph. See algorithms section for information on finding SCCs.
- Multigraph – A graph which can have multiple edges between two vertices, or edges which start and end at the same vertex
- Directed Acyclic Graph (DAG) – A directed graph which has a clearly defined source vertex and sink vertex, where no path exists which goes from a descendent vertex to the source vertex, and has no cycles.
- Linearization – An ordering of the vertices of the graph such that the source vertex is placed first and the sink vertex is placed last
- Sparsity – Defined by how many edges a graph has in relation to its number of vertices. A graph is either dense or sparse
- Dense – A term used to describe a graph which has close to $|V|^2$ edges. (eg. A graph of cities and roads connecting them)
- Sparse – A term used to describe a graph which doesn't have as many edges compared to the number of vertices (eg. The Internet)
- Planar – A term used to describe a graph which has the ability to be represented on a plane without edges crossing each other.
- Dual Graph – A name for a graph which corresponds to a map in a graph coloring problem.
- Tree – A special type of directed graph in which there exists one source vertex connected to generations of child vertices, each with child vertices of their own.
- d-ary Tree – A special type of tree in which each vertex can have up to d outgoing edges. A tree with n vertices has $\log_d n$ generations.

Graphs can be stored in many ways. There are two common representations of a graph.

- Matrix – A matrix where each row and each column represent the vertices of the graph, and each cell has a binary value; 1 if the edge between vertex x and vertex y exists, or 0 if it doesn't. Occupies $O(|V|^2)$ space, but is good for dense graphs.
- Adjacency List – A linked list of vertices, each with a linked list corresponding to the vertices its connected to. Occupies $O(|E|)$ space. This is good for sparse graphs.

HEAPS

A Heap is a data structure which resembles a tree in which the maximum or minimum element is always at the top of the heap, depending on which type of heap it is.

- Min-Heap – Every child node is greater than its parent.
- Max-Heap – Every child node is less than its parent.

The Heap Property: each node is greater than or equal to each of its children according to some comparison predicate which is fixed for the entire data structure.

Procedure	Binary Heap	Binomial Heap	Fibonacci Heap
Make-Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
Extract-Min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
Union	$\Theta(n)$	$O(\log n)$	$\Theta(1)$
Decrease-Key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Delete	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

BINARY HEAP – DPV 114

Elements are stored in a complete binary tree, in which each level is filled from left to right and must be full before the next level is started.

To insert, we place the new element at the bottom of the tree and percolate it up the tree in a method called "`bubbleup`". To remove the minimum (or maximum) element we return the root element, and place the last element in the heap at the top, and let it percolate down in a method called "`siftdown`".

```
Function insert(h,x)  O(log n)
Bubbleup(h,x,|h|+1)

Function decreasekey(h,x)  O(log n)
Bubbleup(h,x,h^-1(x))

Function deletemin(h)  O(log n)
if |h| = 0
  return null
else:
  x = h(1)
  siftdown(h, h(|h|), 1)
  return x

Function makeheap(S)  O(n log n)
H = empty array of size |S|
For every element in S, x:
  H(|H|+1) = x
For I = |S| down to 1:
  Siftdown(h, h(i), i)
Return h
```

These functions are specifically for a binary heap. They can also be found on DPV 123

```
Function bubbleup(h,x,i)  O(log n)
(place x at i and bubble up)
P = ceiling(i/2)
While i != 1
   && key(h(p)) > key(x):
  h(i) = h(p), i = p
  p = ceiling(i/2)
h(i) = x

Function siftdown(h,x,i)  O(log n)
(place x at i and sift down)
c = minchild(h,i)
while c != 0
   && key(h(c)) < key(x):
  h(i) = h(c), i = c
  c = minchild(h,i)
h(i) = x

Function minchild(h,i)  O(log n)
(return the index of the smallest
child of h(i))
if 2i > |h|:
  return 0 (no children)
else:
  return min{key(h(j)) : 2i <= j <=
min{|h|, 2i+1}}
```

'*' means this topic was presented in semester two

d-ARY HEAP – DPV 115

This data structure is just the same as binary heap (as we can substitute "2" for d and we have a binary heap). When we increase the value of d, `insert`s are sped up by a factor of $\Theta(\log d)$, but `deletemin`s are slowed down a bit, by a factor of $\Theta(d \log_d n)$

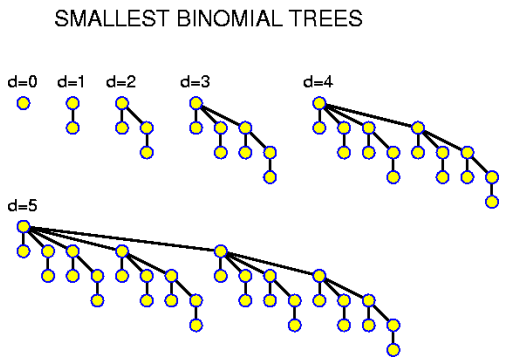
BINOMIAL HEAP – CLRS 455

A binomial heap is a collection of binomial trees – often called a binomial forest. The binomial tree B_k is an ordered tree which is defined recursively. A B_0 tree is a single node. A B_k tree consists of two B_{k-1} trees linked together.

For the binomial tree B_k:

- There are 2^k nodes
- The height of the tree is k
- There are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \ldots, k$
- The root has degree k

See CLRS chapter 19 for more detailed information



FIBONACCI HEAP – CLRS 475

A Fibonacci heap is similar to a binomial heap, but is not constrained to contain only binomial trees. They were only briefly talked about in class, and have better amortized runtime for common operations. Runtimes can be found above in the "Heaps" section. For more detailed information, reference CLRS 475.

AVL TREES

An AVL tree is a self-balancing search tree. It does so by performing so called "rotations" between nodes in the tree which balance the tree during an insertion or a deletion.

Lookup, Insertion, Deletion: $O(\log n)$

'*' means this topic was presented in semester two

*SPLAY TREES

A Splay tree is a self-balancing tree which percolates a value all the way to the root of a tree whenever it is accessed. This ensures that frequently accessed items are easy to reach, and less frequently items take longer to access as a trade-off. In order to percolate an accessed value all the way to the top of the tree, the algorithm uses a rotation algorithm similar to AVL trees. There are three types of rotations

- Zig – Only used for the final rotation to overcome the parity problem,
- Zig-Zig – A Right rotation followed by a right rotation – or a left rotation followed by a left rotation
- Zig-Zag – A Right rotation followed by a left rotation, or vice versa

Each access, the tree "Splay"s itself, in order to bring the accessed value to the top of the tree. Splay is an $O\big((m+n)\log n + m\big)$ algorithm, so it follows that accessing a value takes the same time. Look up and removal have $O(\log n)$ amortized time, however due to the splay functionality

```
Function Splay(x)
  {p(null)=left(null)=right(null)=null}
  While p(x) = null,
    x = left(p(x))
    If g(x) = null, rotate right(p(x))
    If p(x) = left(g(x)), rotate right(g(x)); rotate right(p(x))
    If p(x) = right(g(x)), rotate right(p(x)); rotate left(p(x))
  While( p(x) = null,
    x = right(p(x))
    If g(x) = null, rotate left(p(x))
    If p(x) = right(g(x)), rotate left(g(x); rotate left(p(x))
    If p(x) = left(g(x)), rotate left(p(x)); rotate right(p(x))

Where p(x) gives the parent of x, and g(x) gives the grandparent of x

Function rotateleft(y)
If x, z: x = right(y) and z = p(y)
  If z = null, skip
  If left(z) = y, left(z) = x
  If right(z) = y, right(z) = x
  Left(x), right(y) = y, left(x)
  P(x), p(y) = z, x;
  If right(y) = null, skip
  If right(y) != null, p(right(y)) = y

Function rotateright(y) is symmetric
```

For reference on rotations, see the tree rotation diagram on AVL trees above.

*MINIMUM SPANNING TREES (MST) – DPV 127

A Minimum Spanning Tree (MST) is a subgraph of a graph which connects all vertices together in the minimum number of edges. Greedy algorithms are generally used to find these edge sets. Any sort of an algorithm devised to find MSTs utilizes "The Cut Property".

Algorithms for finding MSTs are Kruskal's algorithm and Prim's algorithm. (See Algorithms Section)

*THE CUT PROPERTY

Suppose edges X are part of a minimum spanning tree of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and $V - S$, and let e be the lightest edge across this partition. Then $X \cup \{e\}$ is part of some MST.

See DPV 128 for a diagram of a MST

*FLOW NETWORKS – DPV 198

Flow networks are graphs which have a source and a sink, along with weighted edges. They are generally used in Max-Flow problems. See DPV 202 and 203 for examples of flow networks.

ALGORITHM DESIGN TECHNIQUES

DIVIDE AND CONQUER

Divide and Conquer algorithms split the problem into a defined number of subproblems, solve the subproblems recursively, then combine the solved subproblems to form a complete solution. In order to design a divide and conquer algorithm, the problem should first be broken up into subproblems of fractional size, and a method should be written to combine solutions.

Examples include the multiplication algorithm, mergesort, median finding, and matrix multiplication.

In many cases, while analyzing runtime of divide and conquer algorithms, we encounter recurrence relations which are telescoping in nature. A handy theorem for finding runtime of divide and conquer algorithms is called the "Master Theorem"

THE MASTER THEOREM

If $T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$ for constants $a > 0, b > 1,\ d \geq 0$, then $T(n) = \begin{cases} O(n^d)\ if\ d > \log_b a \\ O(n^d \log n)\ if\ d = \log_b a \\ O\left(n^{\log_b a}\right)\ if\ d < \log_b a \end{cases}$

In this case, we can assume that are tackling a problem of size n by recursively solving a subproblems of size $\frac{n}{b}$, then combining the answers in $O(n^d)$ time. A proof can be found in DPV 49

*GREEDY ALGORITHMS – DPV 127

Greedy Algorithms are based off of one defining principle – to always do the most immediately satisfying thing. These algorithms may not always provide the optimal solution, but they do get close sometimes.

Examples: Kruskal's Algorithm, Prim's Algorithm, Huffman Encoding, Horn Formulas, Set Cover (approximation)

The main problem with greedy algorithms is that they do not "think ahead" and often produce solutions which are far from optimal due to previous greedy decisions.

*DYNAMIC PROGRAMMING

Dynamic programming is an algorithm design technique which uses a minimization or a maximization in a divide and conquer type approach. Dynamic programming may not always divide the problem into the same size of subproblem (fractional in n),

MISC

AMORTIZED ANALYSIS – CLRS 405

Amortized analysis is the process of finding runtime over a series of repeated operations. For some methods, the effect on the data structure of performing an operation has bearing on future repeats of said operation. In this case, performing an operation which takes $O(p)$, n times may not produce a runtime of $O(np)$ if we perform said operation n times.

POTENTIAL – CLRS 412

The first way of performing amortized analysis is by using a potential based analysis. We can think of a data structure being in a high or low potential state. Using a binary search tree as an example, we can think of the data structure being in a high potential state if it is well balanced, and being in a low potential state if it is not balanced. We can then think of the cost on the data structure by performing an operation as the actual cost of the operation plus the change in potential. If we can define a potential function for a data structure, we can perform a potential based analysis.

The amortized cost of the ith operation with respect to a potential function Φ is defined by:

$$\widehat{c_i} = c_i + \left(\Phi(D_i) - \Phi(D_{i-1})\right)$$

Thus, the total amortized cost over a series of n operations is

$$\sum_{i=1}^{n} \widehat{c_i} = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

Worked out examples can be found on CRLS 414

*ACCOUNTING – CLRS 410

The accounting method of amortized analysis works by assigning differing charges to different operations. We assign the data structure a certain amount of credit, then summing up the costs of n operations and finding the change in credit of the data structure. Amortized cost is equal to the sum of the costs of the n operations. A more detailed description can be found on CLRS 410

*UNIVERSAL HASHING

Universal hashing is a randomized algorithm for selecting a hash function, such that for any two distinct inputs for the generated hash function, x and y, the probability that $F(x) = F(y)$ (ie, a hash collision) is 1/n, n being the number of possible inputs for the function.

Universal hashing is used to generate efficient and functional hash functions for hash tables where nothing can be assumed about the input. A Simple universal class of hash functions is all functions h of the form:

$$h(x) = f\left(g(x)\right), g(x) = \left((ax + b) mod\ p\right)$$

'*' means this topic was presented in semester two

*P / NP

There are two classes of decision problems in the computer science world. A Decision problem is a problem which returns a yes-no answer. Every algorithm discussed in class can be generalized into a decision problem. The two groups of decision problems are

- Those in P, which are solvable in polynomial time
- Those in NP which are checkable in polynomial time

In order for a problem to exist in NP, there simply has to be a polynomial time algorithm to check the solution for accuracy, but in order for that problem to also exist in P, it has to be solved in polynomial time as well.

It is important to note that NP = P, but P is not necessarily equal to NP due to the existence of NP-Complete problems.

*NP COMPLETENESS – DPV 232

NP-Complete problems are a set of problems which cannot be solved in polynomial time (at least that is the current belief). All NP-Complete problems can be reduced to SAT, and a solution to any NP-Complete problem can produce a solution to any other NP-Complete problem. If a solution to an NP-Complete problem can be shown to be polynomial, P = NP and the world explodes.

In order to prove that a problem is NP-Complete, you must reduce a NP-Complete problem to that problem. This process does not work the other way around. For example, SAT can be shown to reduce to 3-SAT (After all, 3-SAT is just a generalization of SAT, and 3-SAT can be reduced to 3D Matching, etc.

*SAT – DPV 232, 260

Given a collection of Boolean clauses in Conjunctive Normal Form (CNF), produce an assignment of all literals which defines each clause to be true. (Proof of NP-Completeness on DPV 260)

*CONJUNCTIVE NORMAL FORM – DPV 234

A Clause which is the disjunction – the logical "OR" – of several literals, or the negation thereof. For example:

$$(x \vee y \vee z), (x \vee !\, y)$$

*3-SAT – DPV 250

3-SAT is SAT, where each clause can only have up to three literals in it. NOTE that if a one literal clause exists, the problem degenerates into a Horn Formula, which is not a NP-Complete Problem. (Reduction found on DPV 250)

*INDEPENDENT SET – DPV 249

Given a graph, find the size of the largest set of vertices which do not touch each other. (Reduction found on DPV 249)

*VERTEX COVER – DPV 252

A special case of Set cover in which we are looking for a collection of vertices such that each edge is touched by a vertex in the set. (Reduction on DPV 252)

*CLIQUE – DPV 252

Given a graph and a goal g, find a set of g vertices such that all possible edges between them are present. (Reduction on DPV 252)

*3D MATCHING – DPV 252

Given a tri-partite graph (think boys, girls, and pets) and a series of edges between the vertices in this graph (think "Boy1 likes Girl2 and Pet3"), find a set of n disjoint edges such that each triple is matched correctly. (Reduction on DPV 252)

*ZOE – DPV 254

Given an $m \times n$ matrix A with 0-1 entries, we must find a 0-1 vector $x = (x_1, \ldots, x_n)$ such that $Ax = 1$ in which we denote a column vector of 1's by "1". (Reduction on DPV 254)

*SUBSET SUM – DPV 255

Given a sequence of integers $A_1, \ldots A_n$, find the maximum value of $\sum_{k=i}^{j} A_k$. Ex. {-1, 11, -4, 13, -5, -2}: 20. (Reduction on DPV 255)

*RUDRATA CYCLE/PATH – DPV 256

Also known as "Hamiltonian Cycle/Path". Given a graph and a starting vertex, find a path which visits every vertex exactly once. Rudrata cycle requires that the path must also end on the starting vertex. (Reduction on DPV 256)

*TRAVELLING SALESMAN PROBLEM – DPV 260

Given a graph with weighted edges and a start vertex, find a path within a budget which visits every vertex once and ends at the start vertex. (Reduction on DPV 260)

*UNSOLVABLE PROBLEMS – DPV 263

There exist problems which are unsolvable. Take for instance, the infinite loop detector `terminates(x,y)` which returns true if x will terminate given a set of inputs y. If we had a solution to this problem, we could set up a function called `paradox` which would loop as long as `terminates` returns true, thus the program would not terminate and it creates a paradox. More info on DPV 263.

FINAL WORDS

BIBLIOGRAPHY

Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani, McGraw-Hill, ISBN 0073523408.

Introduction to Algorithms (2nd Ed) by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, McGraw-Hill, ISBN 0070131511.