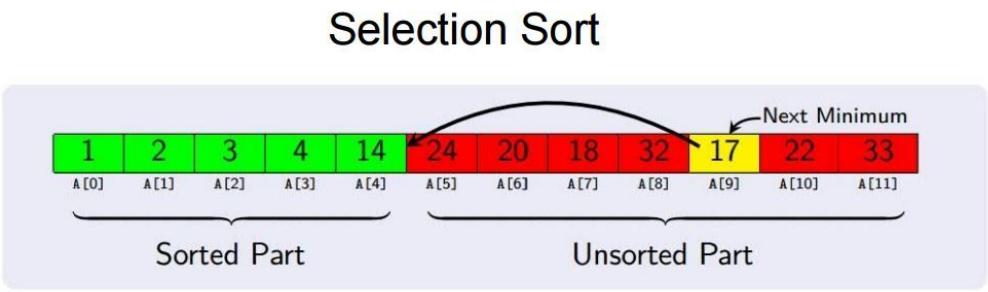
Divide and Conquer

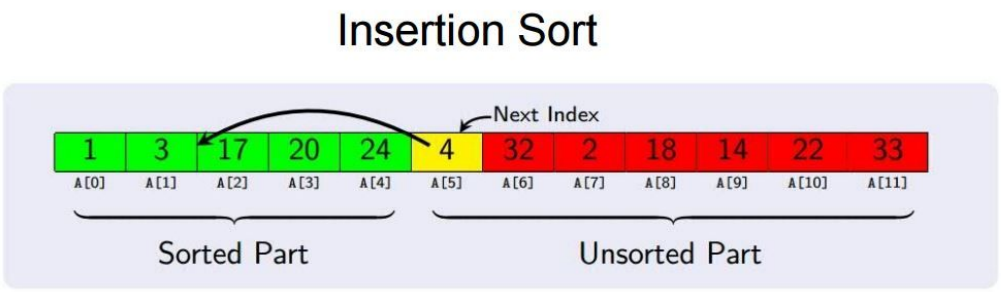
1. Divide problem into smaller parts
2. Solve parts independently
3. Combine solution of parts to produce overall solution



Selection Sort

For(int i = 0; i < n; i++)
findMin(i, n);
swap(i, j);

1. Iterate through and find the minimum
2. Add it to the end of sorted elements
3. Repeat till the end

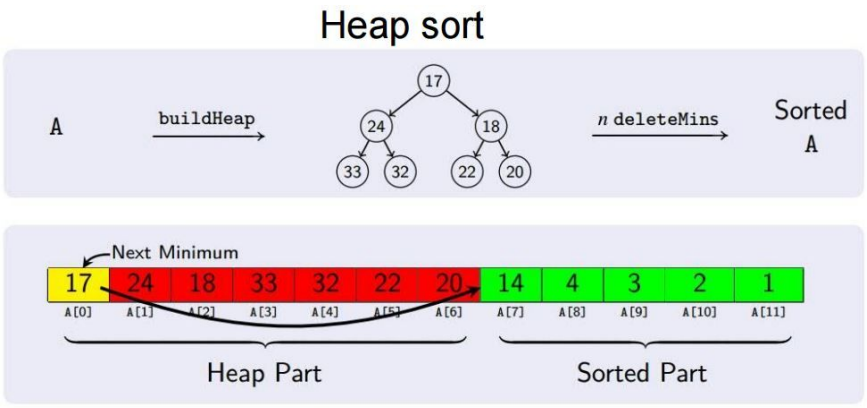


Insertion Sort

O(n^2)

```
for i = 1 to length(A) - 1
   x = A[i]
   j = i
   while j > 0 and A[j-1] > x
      A[j] = A[j-1]
      j = j - 1
   end while
   A[j] = x[3]
end for
```

1. Start from beginning and iterate through
2. If you find a number less than the previous number, insert that number in the correct position of the array
3. Repeat until the end of the array

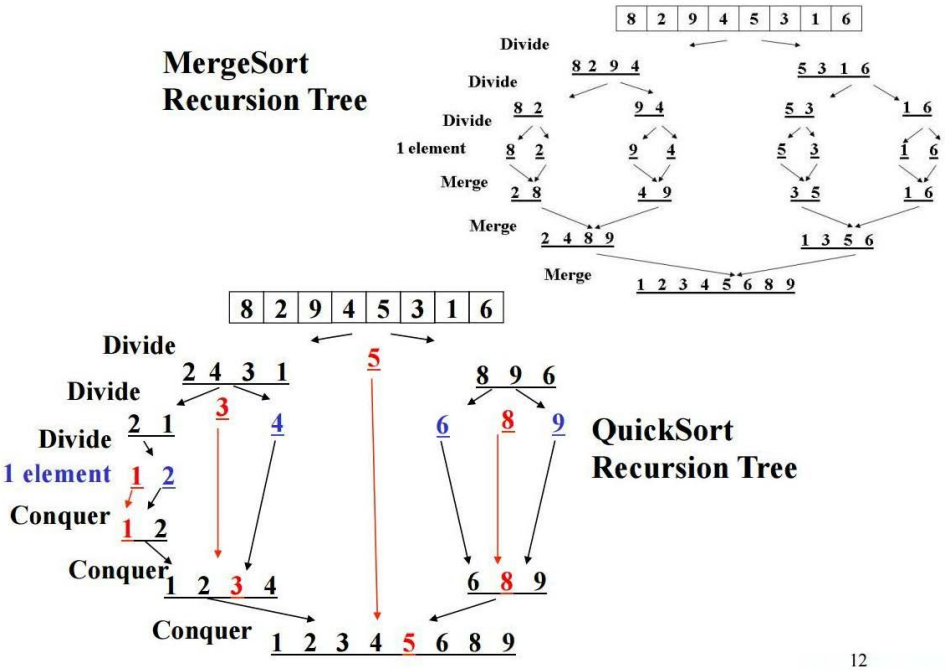


Heap Sort

```
E[] A = buildHeap();
for( i = 0; i< n; i++)
int min = A.deleteMin();
//put min in array
```

1. Call buildMaxHeap() which builds a heap from a list of O(n) operations
2. Swap the first element of the list with the final element
3. Sift the first element to its appropriate index in the heap
4. Repeat step 2 unless the range of the list is one element

Merge Sort



1. Sort the left half of the elements recursively
2. Sort the right half of the elements recursively
3. Merge the two sorted halves into a sorted whole

1. Pick a pivot (most of the time the median)
2. Divide elements into those less than pivot and those greater than the pivot
3. Sort the two divisions recursively

Bucket Sort

count array	
1	3
2	1
3	2
4	2
5	3

- Example:
  K=5
  input (5,1,3,4,3,2,1,1,5,4,5)
  output: 1,1,1,2,3,3,4,4,5,5,5

What is the running time?

1. Set up an array of empty buckets
2. Scatter: Go over the original array, putting each object in its bucket
3. Sort each non-empty bucket
4. Gather: Visit the buckets in order and put all elements back into the array
   * if k is much greater than n, use bucket sort

Radix Sort

Radix = 10

0	1	2	3	4	5	6	7	8	9
	721		3 143				537 67	478 38	9

Input: 478
537
9
721
3
38
143
67

First pass:
1. bucket sort by ones digit
2. Iterate thru and collect into a list
- List is sorted by first digit

Order now: 721
3
143
537
67
478
38
9

Second pass analyzes ten's place and third pass analyzes third the hundred's place

1. Take the least significant digit and group of bits of each key
2. Group the keys based on that digit
3. Repeat the grouping process with each more significant digit

Lower bound on Height

- A binary tree of height h has at most how many leaves? **L ≤ 2 h**
- A binary tree with L leaves has height at least: **h ≥ log2 L**
- The decision tree has how many leaves: **N!**
- So the decision tree has height: **h ≥ log2 N!**]
-

RADIX SORT ALTERNATIVE EXAMPLE

Using the same array from above → 478, 537, 9, 721, 3, 38, 143, 67

Steps:

- **First find the number of N digits in the biggest number**. In this case it is **721** with 3 digits. Now, all the elements in the array need to have the same number of digits as the biggest number.
- We pad temporary zeros in front of each number that does not have N number of digits = biggest number N number of digits. i.e we should assume that there is an imaginary zero in front of this number.
- Resulting array → **478, 537, 009, 721, 003, 038, 143, 067**
- If there are N number of digits in the biggest number the that N is equal to the number of passes. i.e. we will only perform 3 operations in this case.

- Remove temporary zeros once we reach N passes.

478, 537, 009, 721, 003, 038, 143, 067
Create radix size 10, since there are 0-9 digits in a number.

PASS 1

0	1	2	3	4	5	6	7	8	9
	721		003				537	478	009
			143				067	038	

→ 721, 003, 143, 537, 067, 478, 038, 009

PASS 2

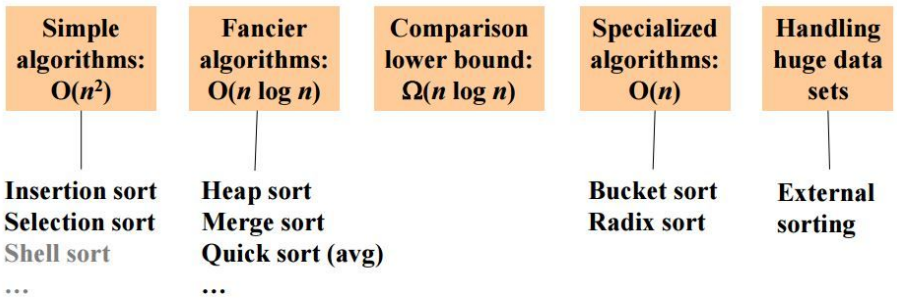
0	1	2	3	4	5	6	7	8	9
003		721	537	143		067	478		
009			038						

→ 003, 009, 721, 537, 038, 143, 067, 478

PASS 3

0	1	2	3	4	5	6	7	8	9
003	143			478	537		721		
009									
038									
067									

→ 003, 009, 038, 067, 143, 478, 537, 721
We reached our final pass. Now remove temporary zeros.
→ 3, 9, 38, 67, 143, 478, 537, 721 (FINAL RESULT IN SORTED ORDER)



External Sorting
Basic Idea:

• Load chunk of data into Memory, sort, store this “run” on disk/tape
• Use the Merge routine from Mergesort to merge runs
• Repeat until you have only one run (one sorted chunk)

For example, for sorting 900 megabytes of data using only 100 megabytes of RAM:

1. Read 100 MB of the data in main memory and sort by some conventional method, like quicksort.
2. Write the sorted data to disk.
3. Repeat steps 1 and 2 until all of the data is in sorted 100MB chunks (there are 900MB / 100MB = 9 chunks), which now need to be merged into one single output file.

4. Read the first 10 MB (= 100MB / (9 chunks + 1)) of each sorted chunk into input buffers in main memory and allocate the remaining 10 MB for an output buffer. (In practice, it might provide better performance to make the output buffer larger and the input buffers slightly smaller.)

In Place: Sorted items occupy the same space as the original items.
- (No copying required, only O(1) extra space if any)

Stable: Items in input with the same value end up in the same order as when they began.

Merge Sort is not in place but stable
Quick Sort is in place but not stable
Heap sort is in place but not stable
Selection sort is in place but is sometimes stable and unstable.

BFS is a queue (First in, first out)
DFS is a stack (Last in, first out)

Loop Run Times

```
for (n = N; n > 0; n /= 2)  ← runs through half the elements of N
{
        for (i = 0; i < n; i++)        ← does the same as above
        {
                DO IT;
        }
}
```

½ of n + ½ of n = O(n)

```
for (i = 1; i < N; i *= 2)
{
        for (j = 0; j < N; j++)
        {
                DO IT;
        }
}
```

O(N log2(n)) based on properties of how log works

```
for (i = 1; i < N; i++)                    ← N
{                                                  *
        for (j = 1; j < i*i; j++)          ← N^2
        {
                if (j%i == 0)              ← covers N cases(try plugging in a number)
                {
                        for (k = 0; k < j; k++) ← N^2
                        {
                                JUST DO IT;
                        }
                }
        }
}
```

therefore N * N^2 * N = O(N^4)

****From midterm 1**

```
x = n
while (x > 0) {
    y = n;
```

```
    while(y > 0) {
        sum++;
        y = y / 2;
    }
    x = x / 2;
}
```

****Anyone remember the runtime? and why?** O(log n^2)?

Heap Operations

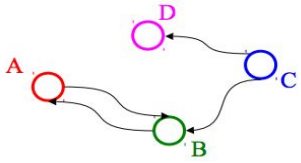
Type	Time Complexity						
	Heapify	Find Max	Extract Max	Increase Key	Insert	Delete	Merge
Linked List (sorted)	-	O(1)	O(1)	O(n)	O(n)	O(1)	O(m+n)
Linked List (unsorted)	-	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Heap	O(n)	O(1)	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(m+n)
Binomial Heap	-	O(1)	O(log(n))	O(log(n))	O(1)	O(log(n))	O(log(n))
Fibonacci Heap	-	O(1)	O(log(n))	O(1)	O(1)	O(log(n))	O(1)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Adjacency Matrix:



	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

Adjacency Matrix Properties

• **Running time to:**

– Get a vertex's out-edges: O(|V|)

– Get a vertex's in-edges: O(|V|)

– Decide if some edge exists: O(1)

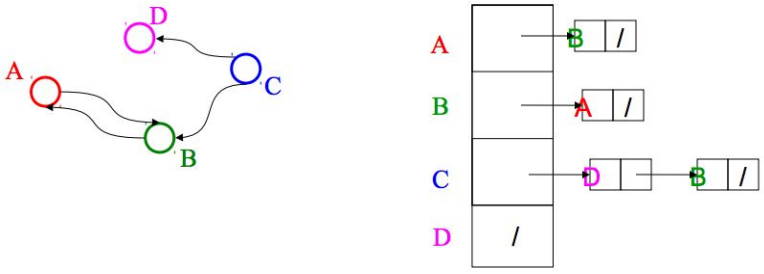
– Insert an edge: O(1)

– Delete an edge: O(1)

● **Space requirements:** |V|2 bits

• **Best for sparse or dense graphs?** Best for dense graphs

Adjacency List:



• Assign each node a number from 0 to |V|-1

• An array of length |V| in which each entry stores a list of all adjacent vertices (e.g., linked list)

Adjacency List Properties

• **Running time to:**

– Get all of a vertex's out-edges: O(d) where d is out-degree of vertex

– Get all of a vertex's in-edges: O(|E|) (but could keep a second adjacency list for this!)

– Decide if some edge exists: O(d) where d is out-degree of source

– Insert an edge: O(1) (unless you need to check if it's there)

– Delete an edge: O(d) where d is out-degree of source

● **Space requirements**: O(|V|+|E|)

• **Best for dense or sparse graphs?** Best for sparse graphs, so usually just stick with linked lists

DAG is a directed graph with no (directed) cycles.

– Every rooted directed tree is a DAG

– But not every DAG is a rooted directed tree

- Recall: In an undirected graph, 0 ≤ |E| < |V|2

• Recall: In a directed graph: 0 ≤ |E| ≤ |V|2

• So for any graph, O(|E|+|V|2) is O(|V|2)

If it is tight, i.e., |E| is (|V|2) we say the graph is **dense.**

If |E| is O(|V|) we say the graph is **sparse.**

Example:

A simple path repeats no vertices, except the first might be the last

[Seattle, Salt Lake City, San Francisco, Dallas]

[Seattle, Salt Lake City, San Francisco, Dallas, Seattle]

• Recall, a cycle is a path that ends where it begins

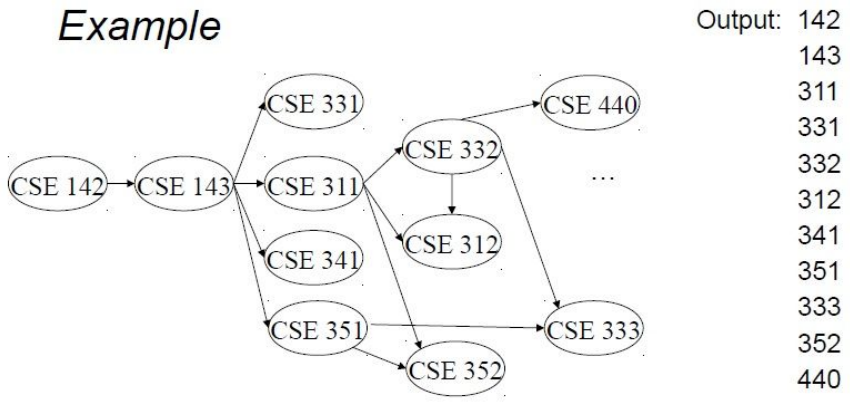
[Seattle, Salt Lake City, San Francisco, Dallas, Seattle]

[Seattle, Salt Lake City, Seattle, Dallas, Seattle]

• A simple cycle is a cycle and a simple path

[Seattle, Salt Lake City, San Francisco, Dallas, Seattle]

Topological Sort



Output: 142 143 311 331 332 312 341 351 333 352 440

Node:	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x	x	x	x	x
In-degree:	0	1	1	2	1	1	2	1	1	2	1
		0	0	1	0	0	1	0	0	1	0
				0			0			0	

1. Label ("mark") each vertex with its in-degree

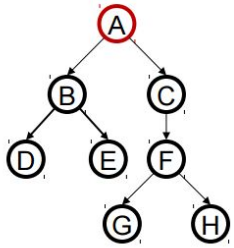
– Think "write in a field in the vertex"

– Could also do this via a data structure (e.g., array) on the side
2. While there are vertices not yet output:
a) Choose a vertex v with labeled with in-degree of 0
b) Output v and conceptually remove it from the graph
c) For each vertex u adjacent to v (i.e. u such that (v,u) in E), decrement the in-degree of u

DFS: (Depth) recursively explore one part before going back to the other parts not yet explored

Example: trees

- A tree is a graph and DFS and BFS are particularly easy to "see"



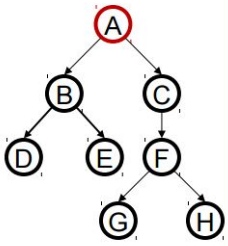

```
DFS(Node start) {
  mark and process start
  for each node u adjacent to start
    if u is not marked
      DFS(u)
}
```

A, B, D, E, C, F, G, H
Exactly what we called a "pre-order traversal" for trees
– The marking is because we support arbitrary graphs and we want to process each node exactly once

BFS: (Breath Search) explores areas closer to the start node first

Example: trees

- A tree is a graph and DFS and BFS are particularly easy to "see"




```
BFS(Node start) {
  initialize queue q to hold start
  mark start as visited
  while(q is not empty) {
    next = q.dequeue() // and "process"
    for each node u adjacent to next
     if(u is not marked)
       mark u and enqueue onto q
  }
}
```

A, B, C, D, E, F, G, H
A "level-order" traversal

Dijkstra's Algorithm

- Node starts at A, locates all nodes around A and continues to further nodes
- It searches through and finds the smallest numbers first
- O(|E|log|V|)

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  build-heap with all nodes
  while(heap is not empty) {
    b = deleteMin()
    b.known = true
    for each edge (b,a) in G
     if(!a.known)
      if(b.cost + weight((b,a)) < a.cost){
        decreaseKey(a,"new cost - old cost")
           a.path = b
      }
}
```

O(|V|)

O(|V|log|V|)

O(|E|log|V|)

O(|V|log|V|+|E|log|V|)

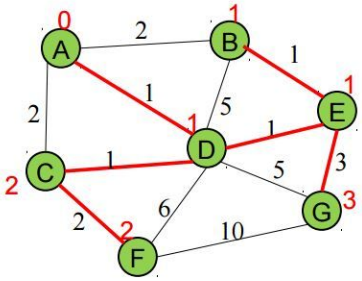
39

Prim's Algorithm

DFS form of Dijkstra's algorithm

- You can access any known node and find whatever element is smallest next to each known node to find the minimum spanning tree
- Cost is distance from one node to the other
- O(|E|log|V|)

-



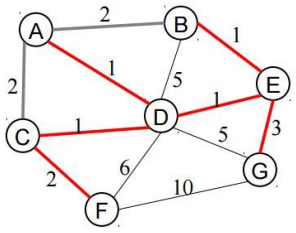
vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G	Y	3	E

39

Kruskal's Algorithm

- Group all numbers together and start with the lowest numbers first and make all nodes known to make the minimum spanning tree
- Skip a edge if it creates a cycle
- Run Time: O(E log(E))

Example



Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

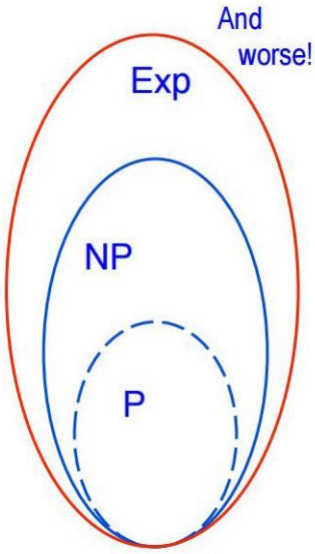
Output: (A,D), (C,D), (B,E), (D,E), (C,F), (E,G)

-

P or NP (Polynomial or NonDeterministic Polynomial)
Decision Problem: Can you answer these problems with a yes/no answer?

- Is there a path from node A to node B with weight < 5? etc.

P = the set of (decision) problems solvable by computers in polynomial time (easy problems)
Examples: sorting, shortest path, MST, Dijkstra's Algorithm



What do we mean by non-deterministically polynomial (NP)?

- On a deterministic machine the next step to execute is unique based on current instruction
- Set of all problems verifiable in polynomial time(wider scale then just p)
- On a Theoretical nondeterministic machine:
- There is a choice of steps, the machine is free to choose any, and it will always choose the correct one leading to solution
- Optimal guesser
- Every problem in P is in NP
- Every problem in NP is in exponential time
- P⊆NP⊆Exp
- We know P ≠ Exp, so either
- P ≠NP, or NP ≠ Exp

NP- Hard: If a problem is NP-Hard, we can reduce any NP problem to that particular NP-Hard problem. But if we can solve an NP-Hard problem in poly time, than we can easily solve an NP problem in poly time.

NP- Complete: set of problems in NP for which it is possible to reduce any other NP problem to polynomial time(hardest problems in np)

- A problem is NP complete if the problem is in NP and if it is reducible to polynomial time

Examples: Longest Path, traveling salesman problem
Exponential = x^N
Examples: Towers of Hanoi

None of what is in NP complete is in Polynomial time
P, NP and NP complete are all verifiable in Polynomial time
NP and NP complete problems may or may not be solvable in P time

Reductions: to "reduce A to B" means to solve A given a subroutine solving B.

- Reducing sort to findMax etc.

We findMax continuously and add to a stack. Once done, the LIFO prints the sorted list.

Greedy Algorithm
An example of a greedy algorithm: Prim's, Kruskal's, Dijkstra's
– At each step, irrevocably does what seems best at that step

- A locally optimal step, not necessarily globally optimal

– Once a vertex is known, it is not revisited

- Turns out to be globally optimal

Divide and Conquer
Ex's: MergeSort, QuickSort,

- Divide a problem into at least two smaller problems.
- The subproblems can (but not necessarily) be solved recursively.
- Conquer by forming the solution to the original problem from the solutions to the smaller problems.

Dynamic Programming
o Break a problem into smaller subproblems.

- But we don't know exactly which subproblems to solve.
- these smaller subproblems complete faster than running the entire problem in whole

o We solve them all and store the results in a table.

- Use a table instead of recursion.

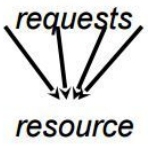
o Use the stored results to solve larger problems.

Parallelism

Sequential programming - everything part of one sequence(what we normally do in programming)

Parallelism:
Use extra resources to solve a problem faster

Concurrency:
Correctly and efficiently manage access to shared resources



Work: how long it would take 1 processor = T1
Span: how long it would take infinity processors = T(n/k)
Speed-up on P processors: T1 / TP
If speed-up is P as we vary P, we call it perfect linear speed-up
– Perfect linear speed-up means doubling P halves running time
– Usually our goal; hard to get in practice

Parallelism is the maximum possible speed-up: T1 / T(n/k)
– At some point, adding processors won't help
– What that point is depends on the span: *parallel algorithms is about dec span without inc work too much*

To get a new thread running:

1. Define a subclass C of java.lang.Thread, overriding run
2. Create an object of class C
3. Call that object's start method

• start sets off a new thread, using run as its "main"

```
class SumArray extends RecursiveTask
{
    int lo; int hi; int[] arr; // arguments
    SumArray(int[] a, int l, int h)
        { … }
    protected Integer compute(){ // return answer
        if(hi – lo < SEQUENTIAL_CUTOFF)
        {
        int ans = 0;
            for(int i=lo; i < hi; i++)
            ans += arr[i];
        return ans;
        }
        else
        {
            int mid = (hi + lo) / 2;
            SumArray left = new SumArray(arr, lo, mid);
            SumArray right= new SumArray(arr, mid, hi);
            left.fork();
            int rightAns = right.compute();
```

```
                        int leftAns = left.join();
                        return leftAns + rightAns;
                    }
                }
}
        static final ForkJoinPool fjPool = new ForkJoinPool();
         int sum(int[] arr){
```

Sequential cutoff: the point on which the program stops creating threads.

Don't subclass Thread	Do subclass RecursiveTask
Don't override run	Do override compute
Do not use an ans field	Do return a V from compute
Don't call start	Do call fork
Don't just call join	Do call join which returns answer
Don't call run to hand-optimize	Do call compute to hand-optimize
Don't have a topmost call to run	Do create a pool and call invoke

Amdahl's Law is a mathematical theorem – Diminishing returns of adding more processors
Running multiple processes at once would be inefficient because each concurrent processor would slow down the progress of the other program running at the same time.

Let the work (time to run on 1 processor) be 1 unit time Let S be the portion of the execution that can't be parallelized Then: T1 = S + (1-S) = 1 Suppose we get perfect linear speedup on the parallel portion Then: TP = S + (1-S)/P So the overall speedup with P processors is (Amdahl's Law): T1 / TP = 1 / (S + (1-S)/P) And the parallelism (infinite processors) is: T1 / T = 1 / S

– Unparallelized parts become a bottleneck very quickly, ie sequential parts
TP = O((T1 / P) + T) – First term dominates for small P, second for large P

Suppose 33% of a program's execution is sequential
– Then a billion processors won't give a speedup over 3
For 256 processors to get at least 100x speedup, we need: 100 1 / (S + (1-S)/256)

- Which means S .0061 (i.e., 99.4% perfectly parallelizable), which is cray :o