Theory of Algorithms

CSC3003S

Introduction

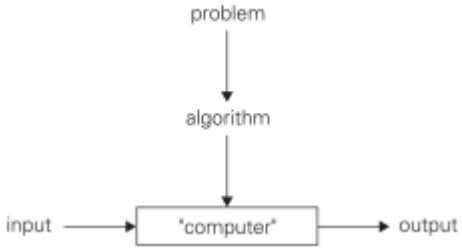
Algorithms

- Algorithm: sequence of unambiguous instructions for solving a well-defined problem
    - Algorithms are guaranteed to terminate if the input is valid
    - A subset of procedures, which aren't guaranteed to terminate
    - Sits above the code level and is realised as code – needs to be unambiguous
        - Otherwise, it can't be converted into code
    - Need to be well-defined, otherwise can't solve it – if there's some uncertainty, then it'll carry over to the instructions represented by the algo
- Can we write an algorithm that given any program as input, will tell us whether or not, for any input, that program stops?
    - Can't prove this – Halting problem

Algorithm Features

- Finite: terminates after a finite number of steps
- Definite: rigorously and unambiguously specified
    - i.e. not possible to take given algorithm and interpret it in different ways so that it provides different behaviour
- Input: valid inputs are clearly specified
    - Can differentiate these from invalid inputs
- Output: can be proved to produce the correct output given a valid input
- Effective: steps are sufficiently simple and basic
    - Can breakdown steps in the algorithm in such a way that it's possible to carry them out

Notion of an Algorithm

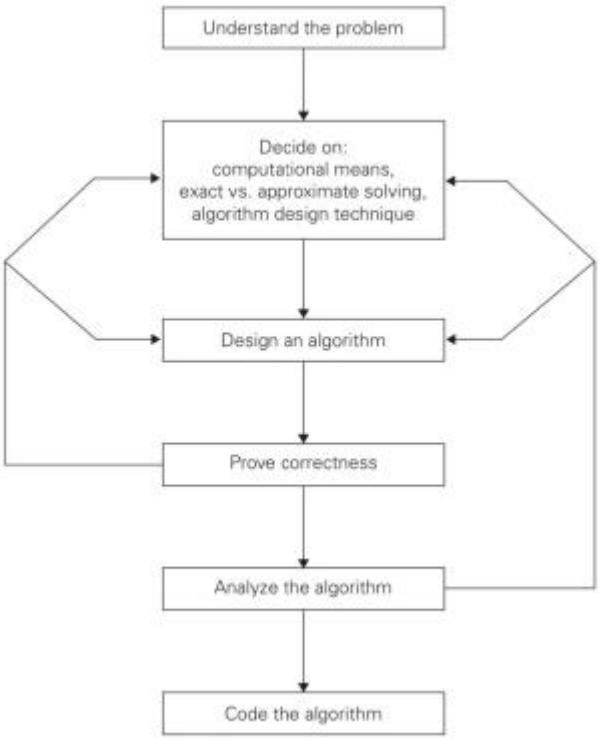


- Diagram shows computer as a black box
    - Note – it's computer in the general sense so could e.g. be an embedded controller, chip on an electronic device, etc
- Each step of the algorithm must be unambiguous
- The range of inputs must be specified carefully
- The same algorithm can be represented in different ways AND
- Several algorithms for solving the same problem may exist - with different properties

Better Algorithms

- BUT – no point in finding fastest algorithm if:
    - That part of the system is not the bottleneck
    - The program will only be run very few times (e.g. reconciling & merging databases when 2 companies merge)
    - Time is not an issue (e.g. can run overnight)

Algorithm Design



- Analyze the Algorithm:
    - Efficiency – time and space
    - Simplicity
    - Generality – range of inputs, special cases
    - Optimality – no other algorithm can do better

Typical Problem Domains

- One of the ways of classifying algorithms is on the basis of the problem itself – because that is what is immediately presented to you
- Sorting and searching
- String Processing
- Graph Problems – structuring the domain as a set of nodes, with links between them
    - E.g. Travelling Salesman
- Combinatorial Problems
    - E.g. List all the permutations of a set
- Geometric Problems

    - Closest-Pair, Convex-Hull
    - GPS: Least number of roads to get from A to B?
- Numerical Problems
    - Solving systems of equations, random numbers, matrix multiplication

Types of Algorithms

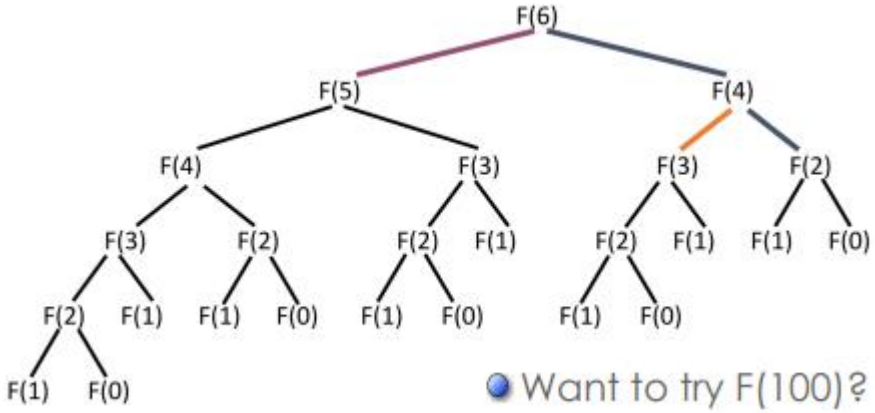
- Brute force
    - Follow definition, try all possibilities
- Decrease & conquer
    - Solve larger instance in terms of smaller instance
- Divide & conquer
    - Break problem into distinct subproblems
- Transformation
    - Convert problem to another one
- Space/Time Tradeoff
    - Use additional data structures
- Dynamic programming
    - Break problem into overlapping subproblems
- Greedy
    - Repeatedly do what is best now

Solving Fibonacci

Brute Force Solution

```
public long fibo(long n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibo(n-1) + fibo(n-2);
}
```

- Recursive solution – seems okay
- Fibonacci F(6) using Brute Force
    - End up with a binary tree structure



Better Solution

Version 1

```
Fibo(n):
     // Computes nth Fibonacci number
     // Input: Non-negative integer n
     // Output: nth Fibonacci number

     F[0] = 0
     F[1] = 1
     for i = 2 to n do:
          F[i] = F[i - 1] + F[i - 2]
     return F[n]
```

- Do it in forward order – work out all elements up to and including the one you need
- Compute nth fib number, progressing from f0 and f1 and working forward
- Have array F – set first and second elements to 0 and 1 respectively
- Then use for loop from third element to the nth element
    - At each step, lookup previous two, add them together and then place in current element
- Better, but might run into some problems

Version 2

```
public BigInteger fibo(int n) {
      if (n==0) return 0; if (n==1) return 1;
      BigInteger [] table = new BigInteger[3];
      table[0] = new BigInteger("0");
      table[1] = new BigInteger("1");

      for (int i=2; i<n; i++) {
            table[2] = table[1].add(table[0]);
            table[0] = table[1];
            table[1] = table[2];
      }
      return table[2];
}
```

- In what ways is this better or worse than v1?

- Using big integer – this is because fib sequence grows exponentially, so use this to prevent integer overflow
- No longer using an entire array – reusing info
  - Just have a 3-element array – because don't need to store all the prior elements in the series, only the last 3
  - Overwrite data in previous elements

- Good space complexity and also solves the problem for large data sizes

By Formula

```
public long fibo(int n) {
     return Math.round(
          (1.0/Math.sqrt(5.0))*
          Math.pow(1.6180339887,n));
}
```

- Choose between elements in the solution space when choosing which algo
- Superficially, looks extremely efficient.
  - But it has practical implementation problems
- Problems with the Formula
  - Using Java It calculates Fibo up to 42 correctly
  - But for Fibo(43) it gives 433494436. The correct answer is 433494437
    - It gets increasingly worse for higher n
  - Floating point rounding problems!

Greatest Common Divisor

- Given two positive integers m and n, find their greatest common denominator
  - That is the largest integer which divides them both without remainder
  - E.g., GCD(60, 24) = 12

Euclid's Solution

```
int gcd(int m, int n) {
     if (n==0) return m;
     return gcd(n, m % n);
}
```

```
gcd(14142, 3131)
= gcd(3131, 1618)
= gcd(1618, 1513)
= gcd(1513, 105)
= gcd(105, 43)
= gcd(43, 19)
= gcd(19, 5)
= gcd(5, 4)
= gcd(4, 1)
= gcd(1, 0)
= 1
```

- Recursive procedure which takes inputs and reduces them down
- Very efficient
- Take two parameters – m and n
  - If one reaches 0, then m represents the GCD – base case of recursive function
- Example with 14142 and 3131 – shows co-primes

- i.e. no GCD > 1 for these inputs

Euclid's Solution without Recursion

```
int gcd(int m, int n) {
     int r;
     while (n != 0) {
          r = m % n;
          m = n;
          n = r;
     }
     return m;
}
```

- Iterative version
- Have an internal variable r – iterate until n becomes 0
  - In the while loop – r is m mod n
  - Assign m to n and r to n
- No more efficient than the recursive solution for some compilers, because the recursive solution is tail recursive
  - i.e. descends rapidly in a straight line as the last call is made to the recursive step
  - A c++, compiler will optimise it and remove the recursion

Version 2

```
int gcd(int m, int n) {
     int t  = min(m,n);
     while(m % t > 0 || n % t > 0) {
          t--;
     }
     return t;
}
```

- Look at m and n – take minimum of the two
  - Check if that is gcd – if not, reduce the value and check repeatedly
- While loop checks if t is common divisor of the two
  - Terminates when t = 1
- GCD(60, 42) tries 42, 41, …

Algorithm Efficiency

- Interest here is working out algo efficiency
  - Computational efficiency – how long an algo takes to run on a certain set of inputs
  - Space efficiency – how much memory does an algo consume over the course of its run
  - These are not entirely independent – consumption of memory can affect run times
  - Overriding concern is measuring runtime performance – need a robust measure which isn't overwhelmed with small, scale fluctuations
- We need measures of:
  - Input's size (typically we call this n)
    - Influences computational efficiency
  - Unit of measuring time – the basic operation of the algorithm
    - Done by looking at operations that are most commonly executed in algo – use that as one unit of execution
- We're usually interested in growth order (broad asymptotic complexity)
  - So, $O(n^2)$ vs $O(n^3)$ is more important than $1412n^2$ vs $4n^3$
- Also interested in best, average and worst performance

Linear Search

- Want to find if a specific element exists in an array – start at beginning of array, stopping when find that point (if its there)
- What is its worst case on array of length n?
  - It looks at all n elements.
  - Its worst-case efficiency is $\Theta(n)$ and $O(n)$
- In the best case it looks at 1 element.
  - Its best-case efficiency is $\Theta(1)$ and $O(1)$
- On average it will look at n / 2 elements before finding the element being searched
  - Its average case efficiency is therefore $\Theta(n / 2)$ or $O(n / 2)$
  - But the 1⁄2 is a constant, so it's $\Theta(n)$ or $O(n)$

Typical Efficiency Classes

n	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n	n!
10	3.3	10	33.2	1E+02	1E+03	1E+03	4E+06
100	6.6	100	664.4	1E+04	1E+06	1E+30	9E+157
1,000	10.0	1,000	9,965.8	1E+06	1E+09	1E+301	
10,000	13.3	10,000	132,877.1	1E+08	1E+12		
100,000	16.6	100,000	1,660,964.0	1E+10	1E+15		
1,000,000	19.9	1,000,000	19,931,568.6	1E+12	1E+18		

- Have n inputs
- Comparing the number of operations for different input sizes
- Best efficiency class is logarithmic – $\log_2 n$
- Then linear – in proportion to input size
- Log-linear – what sort algos typically fit into
- Evil classes: n^2 and n^3
- Worst: exponential runtime algorithms
    - 2^n and n!
    - Avoid at all costs!

Big O

Intuitively

- Example 1:
    - Algorithm executes: 2n + 10 operations
    - We are not interested in the 2 or the 10
    - This is O(n)
- Example 2:
    - Algorithm executes: $3n^2 + 9n + 5$ operations
    - We are not interested in the 3, 9n or 5
    - This is $O(n^2)$
- How Long to Sort 100 million Items?
    - $n * \log_2 n$ = 2,657,542,476 (in 30s)
    - n^2 = 10,000,000,000,000,000
    - 10^{16} / 2,657,542,476 = 3,762,875
    - 3,762,875 x 30 seconds = 112,886,248s
    - = 3.5 years
    - Insertion sort and Bubblesort are impractical for large data sets!

Definition of Big O

- A function t(n) ∈ O(g(n)) iff (if and only if) there is a c and an n_0 such that t(n) <= cg(n) for all n>=n_0
    - Function on the number of operations and n (input size) – t
    - Belongs to class Big O of some other function if and only if can determine some constant c and input size n_0 so that this function t will always be less than the constant x function of big O for all inputs beyond n_0
    - Upper bound on algo performance
- Big o is not necessarily a tight upper-bound
    - Example 2
- Example 1:
    - 100n+5 ∈ O(n)
    - To see that this is the case set c to 101 and n_0 to 6.
    - Then we must prove that 100n+5 <= 101n for all n>=6.
        - Simple to prove with induction
- Example 2:
    - 100n + 5 ∈ O(n^2)
    - Set c = 21 and n0 = 5
    - So, prove that 100n + 5 <= 21n^2 if n > 5

Definition of Omega Ω

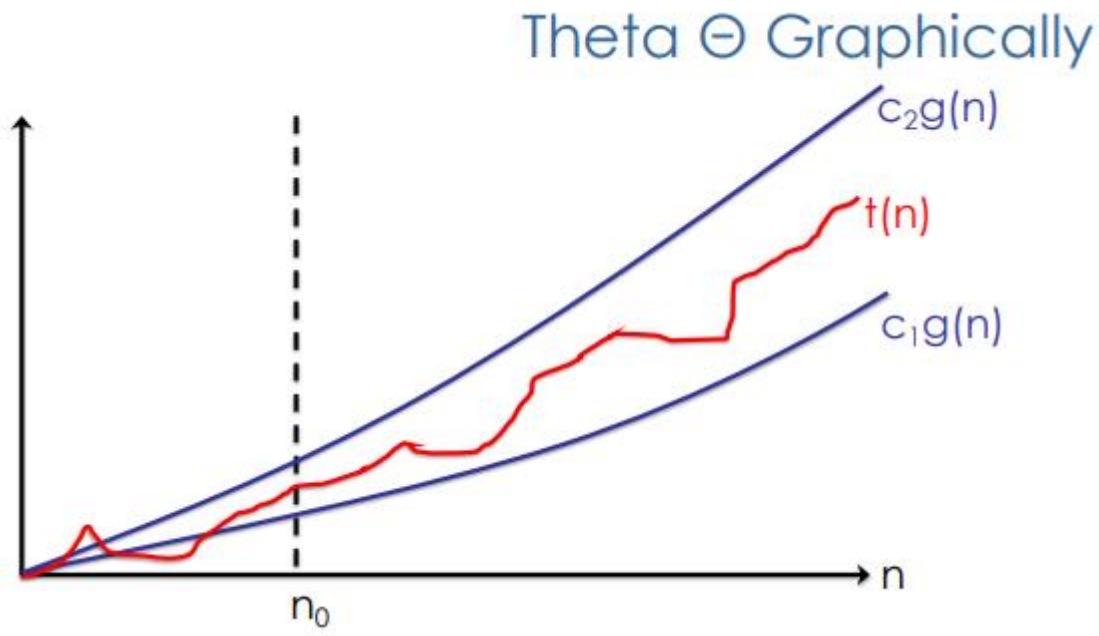
- Lower bound
- A function t(n) ∈ Ω(g(n)) iff there is a positive constant c and a non-negative integer n_0, such that t(n) >= cg(n) for all n>=n_0
- Not widely used

Theta Θ

- Tight bound – both upper and lower – on computational complexity
- Consider sequentially summing an array
    - Its efficiency class is O(n)
    - But it is also by our definition O(n^2) and O(2^n) and O(n log n)
        - i.e. any of the more expensive efficiency classes also subsume the less expensive ones
    - NB: It is not O (log n) – because it is lower than computational cost
- Big O (n^2) means the algorithm's basic operations execute in proportion to n^2 or better
- How do we say that summing array's basic operation executes EXACTLY proportional to n? – Theta Θ
    - Efficiency class for summing an array is Θ (n)
    - It is not: Θ (n^2) or Θ (n log n)
    - It is precisely Θ (n)

Definition of Theta Θ

- A function t(n) ∈ Θ(g(n)) iff there is a c_1, c_2, and n_0 such that c2g(n) <= t(n) <= c1g(n) for all n>=n_0
    - Take an upper and lower bound and "sandwich" function between them
    - Big theta of some function applies if and only if we can choose some constants c1 and c2 and some barrier n0 on the size of the inputs
        - Such that function is less than c1 x big theta and greater than/equal to c2 x big theta
- Same as big O, except functions in this class cannot be in a more efficient class
    - 100n + 5 ∈ Θ(n)
    - 100n + 5 ∈ O(n^2)
    - But 100n + 5 is NOT ∈ Θ(n^2)



- Assume that n is on the x axis and runtime on the y axis
- Can choose some value of n, n_0, for which the function is sandwiched between two layers of the big theta
    - Before n^0, function can vary outside the bounds – but after n_0 mark, it must become sandwiched between them
    - Asymptotic computational complexity classes

Comparing Big O, Omega Ω, Theta Θ

- Big O
    - Running time as n gets larger is at most proportional to g(n)
    - Upper bound
- Omega Ω
    - Running time as n gets larger is at least proportional to g(n)
    - Lower bound
- Theta Θ

- Running time as n gets larger is exactly proportional to g(n)
- Exact bound

Examples

- Average of $O(n^2)$ – algorithm (runtime) grows at most as fast as n2 with average case input
  - E.g., Bubble sort
- Worst case of $O(n^3)$ – algorithm grows at most as fast as n3 with its worst case
  - E.g., brute force matrix multiplication, which is also $\Theta(n^3)$
- Best case of $\Theta(n)$ – algorithm grows linearly in the best case
  - E.g., sum an array
- Worst case of $\Omega(2^n)$ – algorithm grows at best exponentially in worst case
  - E.g., create the power set

Checkpoint

Checkpoint

- True or false: $\Theta(n + \log n) = \Theta(n)$
- True or false: $O(n + \log n) = O(n)$
- True or false: $\Theta(n \log_2 n) = \Theta(n \log_{10} n)$
- True or false: $\Theta(\log^2 n) = \Theta(\log n)$
- True or false: $O(n \log n) = O(n)$
- True or false: if $x \in O(n \log n)$ then $x \in O(n^2)$
- True or false: if $x \in \Theta(n \log n)$ then $x \in \Theta(n^2)$
- True or false: if $x \in O(n \log n)$ then $x \in O(n)$

Checkpoint Solutions

- True: $\Theta(n + \log n) = \Theta(n)$
  - We can discard addition terms that are less significant
- True: $O(n + \log n) = O(n)$
  - Same as above
- True: $\Theta(n \log_2 n) = \Theta(n \log_{10} n)$
  - $\log_2 n / \log_{10} n$ = a constant = 3.32, approximately
- False: $\Theta(\log^2 n) = \Theta(\log n)$
  - $\Theta(\log^2 n)$ grows more than a constant faster than $\Theta(\log n)$
- False: $O(n \log n) = O(n)$
  - These are two different sets. But $O(n)$ is a subset of $O(n \log n)$

Proof that Θ (n + log n) = Θ (n)

- We know that Θ(2n) = Θ(n)
- We also know:
  - n + log n < n + n (because log n < n)
    => n + log n < 2n
    => Θ (n + log n) is **not a worse** efficiency class Θ (n)
- Θ (n + log n) is clearly **not a better** efficiency class than Θ (n) (because n + log n > n)
- So Θ (n + log n) must be the same efficiency class as Θ (n)

Efficiency Derivation - Sequential

Steps for Calculating Efficiency

- These steps apply to both sequential and recursive algos
- (1) Identify Algorithm's basic operations:
  - The operations that are executed repeatedly at the core of the algorithm
  - E.g., comparisons and swapping in sorting
  - E.g., multiplications and additions in matrix multiplication
  - If have choice of several – choose the most expensive one
- (2) Set up an equation which counts the number of basic operations for a given input of size n
  - C(n) = ….
  - In the sequential case – this will involve summation
- (3) Solve the equation for the number of occurrences
  - Turn that into a big O, theta or omega representation

Maths Tools

- S1 $\sum_{i=j}^{n} 1 = n - j + 1$
- R2 $\sum_{i=1}^{u} ca_i = c\sum_{i=1}^{u} a_i$
- R2 $\sum_{i=1}^{u} a_i \pm b_i = \sum_{i=1}^{u} a_i \pm \sum_{i=1}^{u} b_i$
- S2 $\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = n(n+1)/2 \approx n^2 \in \theta(n^2)$
  - i.e., 1+2+3+…+n = n(n+1)/2

- For simple, sequential efficiency calculations
- S1: for a standard for loop, goes from standard counter at index j to n
  - Operation inside the loop costs 1 unit
  - Then # of occurrences is n – j + 1
    - i.e. upper limit – lower limit, subtract 1
- R2: have a summation of c x ai where ai depends on index i
  - Extract c
- R3: can separate out summations

- S2: have summation where amount of work done depends on the index
    - Use sum of geometric sequence – which is equivalent to n^2

Analysis Exercise

Analysis Exercise 1

```
int MysteryFunction( A [0 .. n-1] )
     MysteryVal = A[0]
     for i = 1 to n - 1:
          if A[i] > MysteryVal:
               MysteryVal = A[i]
     return MysteryVal
```

- What does this algorithm do?
- What is its basic operation?
- In terms of n, how many times is its basic operation executed?
- What is its asymptotic efficiency class (Big O)?

Analysis Solution 1

```
int maxElement( A [0 .. n-1] )
     maxVal = A[0]
     for i = 1 to n - 1:
          if A[i] > maxVal :
               maxVal = A[i]
     return maxVal
```

- What does this algorithm do?
    - Finds the largest element in an array
- What is its basic operation?
    - Have the option of either the comparison of the assignment of maxval – but note that the comparison happens for each iteration, but the assignment does not
    - A[i] > maxVal
- In terms of n, how many times is its basic operation executed?
    - $C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \theta(n)$
    - Iterate from 1 to n -1, each time carry out one comparison
    - Solve this equation using S1 – take the upper limit, subtract the lower limit from that and add 1
    - Get n – 1 iterations of the for loop
- What is its asymptotic efficiency class (Big O)? – O(n)/linear

Analysis Exercise 2: Is it a set

- You have loaded a file of strings into an array
- You want to check if there are any duplicates – duplicates aren't necessarily in sequential index positions
- In other words, is the array, A, a set?
    - i.e., each element appears at most once
- Come up with two or three, algorithms to determine if A is a set

- What are their time complexities?
- Which is more efficient?

Brute Force Exercise

```
boolean isset(A)  {
  for (i=0; i<A.length-1; i++)
    for (j=i+1; j<A.length; j++)
      if (A[i] == A[j]) return false;
  return true;
}
```

- Take first element, compare to all other elements to see if any are the same, move onto the second element if no duplicates found, compare to the others in the remainder of the list, and so on
    - Exit early with a false if duplicate found
- What is the worst case?
- What are the core operations?
- How many times does the inner loop execute?

Brute Force Solution

- What is the worst case? – No early out, completion of both nested loops
- What are the core operations? – comparisons
- How many times does the inner loop execute?
    - n-1 + n-2 + n-3 … + 1 times = n(n-1)/2
        - Formula of geometric sequence
        - i is iterated from 0 to n, and the innerloop is iterated from i+1
    - Worst case: $\Theta(n^2)$, average case too

Brute Force Full Derivation

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

- S1 on the second summation and simplifying

$$\sum_{i=0}^{n-2} (n-1-i)$$

- Using R2 this simplifies to

$$\sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$

- Using R1 on the first summation and S2 on the second

$$(n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

- Using S1 on the first summation and simplifying

$$(n-1)^2 - \frac{(n-2)(n-1)}{2} = (n-1)n/2 \approx \frac{1}{2}n^2 \in \theta(n^2)$$

- Have function for computation cost – double summation with single operation
- S1 – use indices of sigma to calculate the # of repetitions

- n-1-(i+1)+1 = n-1-i
- R2 – split into two components
- R1 – can take out n-1 as a constant since it doesn't depend on the indices of sigma
  - Second sigma *does* depend on i though
- S2 – take upper bound and multiple that by (upper bound + 1) and divide both by 2
- S1 – use again on the first summation to get final operation count

Decrease & Conquer

```
1 Boolean isset(Array A, int index=0) {
2   if (index >= A.length-1) return true;
3   for (i=index+1; i<A.length; i++) {
4     if (A[index]==A[i]) return false;
5   }
6   return isset(A, index+1);
7 }
```

- This algorithm is materially identical to the brute force one
  - Each time reduce the problem size by one step
- Worst case: $\Theta(n^2)$. Average case too
- Line 2 – base case of recursion
  - Checks if reached the end of the outer for loop i.e. have we reached the end of the array
  - If it has reached that point and haven't reached a repetition of elements, return true
- Line 3 – otherwise, process all the indices that are left in the array using a for loop
  - For each of those, compare it to the index position provided as an argument to the function
- Line 4 – if encounter a comparison where two elements are the same, short circuit and return false
- Line 6 – recursive step, increment index by 1

Transform & Conquer

```
boolean isset(A) {
  sort(A);
  for (int i=0; i<A.length-1; i++)
    if (A[i] == A[i+1]) return false;
  return true;
}
```

- Here we sort the array and then do a linear search through it
  - Only a single pass required

Comparisons

Brute Force vs. Transform & Conquer

- What is the efficiency of the transform & conquer solution?
  - The most efficient sorting algorithms are $\Theta(n \log n)$
  - The for loop is linear for the worst case: $\Theta(n)$
  - Combined efficiency costs: it runs the sort, then the for loop so this is $\Theta(n \log n + n)$

    - The worst-case order of an algorithm is the efficiency of its worst part – drop the lower order terms
- So the Transform & Conquest algorithm is $\Theta(n \log n)$ which is faster than the brute force solution of $\Theta(n^2)$

Efficiency Derivation – Recursive

Analysing Recursive Algorithms

- Recursive efficiency follows the same pattern as sequential:
  - Determine core operation (what happens most frequently) ⇢ create equation for number of repetitions ⇢ solve ⇢ derive efficiency
- Except that equations are recurrence relations and not summations
  - Need to build a similar maths toolset for recurrence relations
  - Either solve by Backward Substitution or using the Master Theorem

Recurrence Relations

- A recurrence relation is a recursive mathematical function
  - Has two components – recursion step and base case
  - Example – function M(n)
    - Recursive step is where we call the function on a reduced version of itself

E.g., $M(n) = \begin{cases} 0 & if\ n = 0 \\ M(n-1) + 1 & otherwise \end{cases}$

Has a recursive and base case

- A tool for analysing recursive algorithms
  - i.e. a useful representation of the # of times an operation is carried out in an algo
- We will consider these recurrence relations:
  - T(n) = aT(n-k) + f(n) // Like M(n) above OR
  - T(n) = aT(n/b) + f(n)

Analysis Exercise 3: Factorial

```
int F(n) {
     If n == 0 return 1;
     return F(n-1) * n;
}
```

- Basic operation we want to count? – Multiplication
- How many multiplications? – Once for every recursive call
  - Need to know the # to derive the efficiency
- Set a recurrence relation that depicts the # of multiplications

- M(n) = 0 when n =0
  M(n-1)+1 otherwise
- This needs to be solved by method of backwards substitution

Analysis Solution 3

- Let M(n) be number of multiplications
- Then M(0) = 0

  And M(n) = M(n-1) + 1
  M(n) = M(n-1) +1 = [M(n-2) + 1] + 1
  = [M(n-3) + 1] + 2 = M(n-3) + 3
- And in general:
  M(n) = M(n-k) + k
  M(n) = M(n-n) + n = M(0) + n = n

- Take M(n-1) and expand it – see if any pattern develops
- So for the general case have M(n) = M(n-k)+k
  - Solve that for k = n
  - Thus, have n multiplications
- Efficiency is thus Θ(n)

Analysis Exercise 4: Counting Bits

```
int CountBits(int n such that n > 0):
    if n == 1 return 1
    else return 1 + CountBits(n / 2)
```

- Provide a # n such that n>0
- Base case is n=1 – return 1
  - Otherwise recursively call the function with n/2
- Halving the size of the problem on each step
- How do we determine the number of basic operations?
  - Basic op is addition of 1 on each call to CountBits

Analysis Solution 4

- $A(1) = 0$ The addition doesn't take place when n=1
  $A(n) = A(n / 2) + 1$ for $n > 1$
- Now for a clever trick:
  - Let $n = 2^k$ which is the same as saying $k = \log_2 n$
  - Now $n / 2 = ½ \times 2^k = 2^{-1}.2^k = 2^{k-1}$
- $A(1) = A(2^0) = 0$
  $A(n) = A(2^k) = A(2^{k-1}) + 1$ for $k > 0$
  $= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2$
  $= A(2^{k-k}) + k = A(2^0) + k = k = \log_2 n \in \Theta(\log n)$

- Construct recursive recurrence relation – determine base case (no addition), which takes place when n = 1
  - Then the recursive step – divide problem size by 2, add 1 for the addition step that takes place
- Use a trick which assumes that you have a power of two in terms of input size
- Setup recurrence relations in terms of 2^{k-1}
  - Change the recurrence step to match
  - Obtain the pattern – have k as final number of additions
  - Sub in the value of l
- Gives Θ(log n)

Fibonacci Analysis

```
int F(n):
        if n<=1 return n
        else return F(n-1) + F(n-2)
```

- Base case – return either 0 or 1
  - Otherwise return the sum of the previous two values in the sequence
- Basic operation is an addition
- Recurrence relation:
  - For any function of n, # of additions is what is done for A(n-1) + 1 (for the addition itself) + A(n-2)

$A(n) = A(n-1) + 1 + A(n-2)$
Requires a different solution method
$A(n) \in \Theta(1.61803^n)$

Master Theorem

Motivation

- No general solution to all recurrence relations
- But the Master Theorem shows the asymptotic efficiency class (i.e. theta or big O) of recurrence relations of the form:
    - $T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$
    - i.e. recurrence relation with one single recurrent step, some parameter a and some function of n that we add to (the cost of the step itself)
    - Have a direct solution
- Provides a short cut that replaces backward substitution

Theorem

If we have a recurrence of this form:

- $T(n) = aT(n/b) + f(n)$, $f(n) \in \Theta(n^d)$ and $T(1) = c$ then
  - $T(n) \in \Theta(n^d)$ if $a < b^d$
  - $T(n) \in \Theta(n^d \log n)$ if $a = b^d$
  - $T(n) \in \Theta(n^{\log_b a})$ if $a > b^d$
- Analogous results for O and Ω

- Given a recurrence relation with this form, with variables a, b, d and the fact that base case and constant c
- Then, can compare and consider a, b and d
    - Use that as a selection mechanism for Θ
    - E.g. if $a < b^d$, then Θ is proportional to $\Theta(n^d)$

Example 1

- Consider: $A(n) = A(n / 2) + 1$, $A(1) = 0$
  - For pattern: $T(n) = aT(n/b) + f(n)$, $f(n) \in \Theta(n^d)$ and $T(1) = c$
  - $a = 1$, $b = 2$, $c = 0$, and $f(n) \in \Theta(1) = \Theta(n^0)$
  - Thus, $d = 0$
- Select Master Theorem Case:
  - $b^d = 1$
  - Therefore $a = b^d$
  - By Master Theorem: $T(n) \in \Theta(n^d \log n)$
- And since $d = 0$, $T(n) \in \Theta(\log n)$

- Consider that recurrence relation has the form of halving the problem size
    - Sub in the values for a, b, c and d
- Select the correct instance of the master theorem
- Have the same result as using backwards substitution, but a lot faster

Example 2

- Consider: $A(n) = 2\,A(n/2) + 1,\ A(1) = 1$
  - For pattern $T(n) = aT(n/b) + f(n)$, $f(n) \in \Theta(n^d)$ and $T(1) = c$
  - $a = 2, b = 2, c = 1$
  - and $f(n) \in \Theta(1) = \Theta(n^0)$, so $d = 0$
  - Now $2 > 2^0$ so $a > b^d$
  - By master theorem $A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$

- Same as before, just use substitutions

Brute Force

Introduction

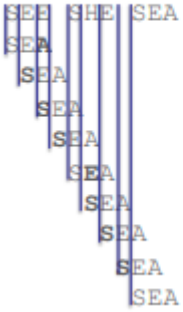
- Brute Force – A straightforward approach directly based on the problem statement
  - Implies lack of sophistication and thought
  - Exhaustive search is a special case
- Nevertheless:
  - Often the quickest and easiest to implement
  - Sometimes the best solution too!
  - More importantly, often it's good enough for occasional use or small problem sizes
- Example: exponentiation an = a * a * a * … * a (n times) and linear search
  - Exponentiation is O(n) – not the best approach

String Matching

- A kind of linear search
- Write a brute force solution to this problem:
  - Given a text, T (string of alphabetic characters), and a pattern string, P, find the index of the first occurrence in the text of P
  - If not found, return -1
- What's the worst-case efficiency class?
- Is the worst case likely?
- Think of a text and string where worst case occurs

Example

Brute force match "SEA" in this text



- Start at the beginning – check characters in pattern against first, second and third position in the text string
  - As soon as fail to match, shift pattern once place to the right and repeat the checke
- With string "see she sea"
  - First check "see" against sea – fails, shift again to one place

Algorithm (High Level)

```
1.Align pattern at beginning of text
2.Moving left to right, compare each character of
  pattern to the corresponding character in text UNTIL
   • All characters are found to match (successful
     search); or
   • A mismatch is detected
3.WHILE pattern is not found and the text is not yet
  exhausted, realign pattern one position to the right
  and repeat step 2.
```

- n characters in Text (T) and m characters in Pattern (P)
- Last viable position is at n-m

Algorithm (Low Level)

```
BruteForceStringMatch ( T[0..n-1], P[0..m-1] )
    // T is the text; P is the pattern we're searching
    // for in the text

    for k ← 0 to n - m do // for each char in T
        j ← 0
        while j < m and P[j] = T[i+j] do // for each char in P
            j ← j + 1
        if j = m return k
    return -1
```

- Take text (size n) and pattern (size m)
- Iterate through the text from 0 to n-m (last possible index position pattern could occur in)
    - J is a local index that passes across the pattern
- While – pattern checks
    - Increment so long as we have not reached m (end of pattern) and still successfully matching pattern against the text
    - If reach the end of the pattern, return current index in text
- Otherwise return -1

Worst Case

- Worst case: the search string matches on every character except the last, for every iteration of the outer loop
    - E.g., Text = "aaaaaaaaaaaaaaaaaaaaaa...." (n chars)
    - Search string = "aaab" (m chars = # of comparisons for the pattern)
        - Only failing on the b each time
        - What is the amount of shifts needed to do across the text? – (n-m+1)
    - = m(n-m+1) character comparisons
        - = $nm-m^2+m$ – as long as n >> m, can remove the lower order terms ($-m^2+m$)
    - = Θ(mn) for m much smaller than n (which is what happens in practice)
- Worst case very unlikely with natural language!
- Average case on natural language? – Θ(n)
- Predictive text!

Closest Pair

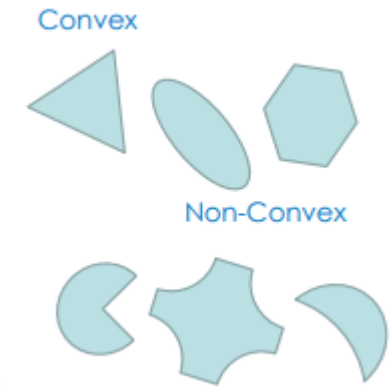
- Problem:
  - Find the two points that are closest together in a set of n 2D points $P_1 = (x1,y1)$, …, $P_n = (x_n, y_n)$
  - Reminiscent of brute-force algorithm for "is this a set"
    - Take a given point and compare it against all the other points – recording the shortest distance
    - Then discard that point, take another and do the same as above

Algorithm

```
dmin ← ∞
     for i ← 1 to n-1 do
          for j ← i+1 to n do
               d ← sqrt((xi - xj)^2 + (yi - yj)^2)
               if d < dmin
                    dmin = d
                    index1 = i
                    index2 = j
return index1, index2
```

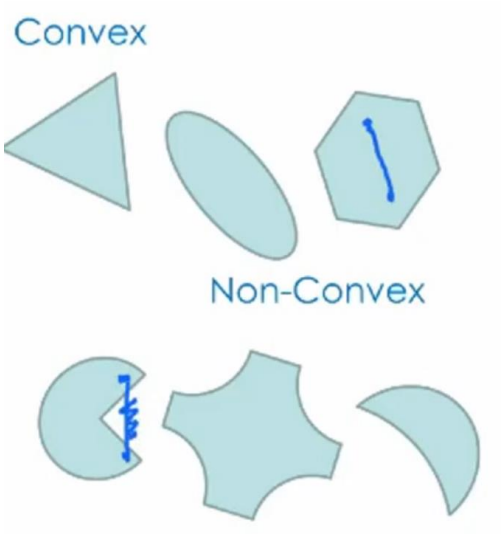
- Set initial distance between points to be ∞
- Iterate over points from 1 to end-1
  - Then iterate from current point+1 to the end
- Look at two points – record the distance between then using Euclidean formula for distance between points
  - If value is less than current running min – adjust, and record position of two points
- End – return the indices
- Efficiency: $\Theta(n^2)$
- Sqrt evaluation is expensive in practice but can be avoided

Convex Hull

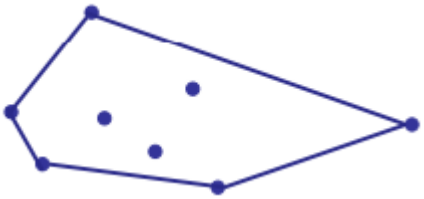


- Problem: find the convex hull enclosing n 2D points
- Convex Hull: if S is a set of points then the Convex Hull of S is the smallest convex set containing S
  - Basically, the smallest enclosing boundary around a set of points
- Convex Set: a set of points in the plane is convex if for any two points P and Q, the line segment joining P and Q belongs to the set

- Have two points within a set – if draw straight line between them, there's no point on that straight line that strays outside the convex set



Algorithm



- For each pair of points p1 and p2 – draw a line connecting them
- Determine whether all other points lie to the same side of the straight line through p1 and p2
- They then form part of the convex hull boundary
- Efficiency: $\Theta(n^3)$
  - Have to get every single pair of points – $O(n^2)$
  - Then have to compare those two points against all other remaining points – O(n)

Pros and Cons of Brute Force

- Strengths:
  - Wide applicability
  - Simplicity Yields reasonable algorithms for some important problems and standard algorithms for simple computational tasks
  - A good yardstick for better algorithms
  - Sometimes doing better is not worth the bother
- Weaknesses:
  - Rarely produces efficient algorithms
  - Some brute force algorithms are infeasibly slow

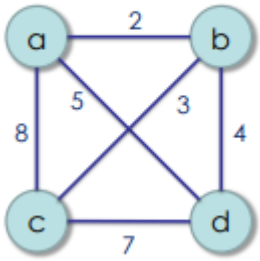
Exhaustive Search

Introduction

- Have a lot of different situations and have to enumerate through all of them in order to determine which is the best
- Definition: a brute force solution to the search for an element with a special property
    - Usually among combinatorial objects such a permutations or subsets
        - Subsets – order within the subset doesn't matter
    - Suggests generating each and every element of the problem's domain
- Method:
    - 1. Construct a way of listing all potential solutions to the problem in a systematic manner
        - All solutions are eventually listed
        - No solution is repeated
    - 2. Evaluate solutions one by one (disqualifying infeasible ones) keeping track of the best one found so far
    - 3. When search ends, announce the winner

Travelling Salesman Problem

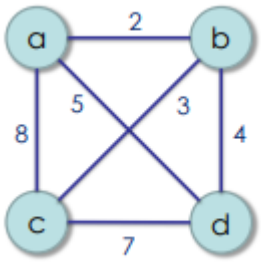
Example



- Problem:
- Given n cities with known distances between each pair
- Find the shortest tour that passes through all the cities exactly once before returning to the starting city
- Alternatively:
    - Find shortest Hamiltonian Circuit in a weighted connected graph

TSP by Exhaustive Search

Tour	Cost
a→b→c→d→a	2 + 3 + 7 + 5 = 17
a→b→d→c→a	2 + 4 + 7 + 8 = 21
a→c→b→d→a	8 + 3 + 4 + 5 = 20
a→c→d→b→a	8 + 7 + 4 + 2 = 21
a→d→b→c→a	5 + 4 + 3 + 8 = 20
a→d→c→b→a	5 + 7 + 3 + 2 = 17



- All permutations of routes
- Improvements:
  - Start and end at one particular city
  - Remove tours that differ only in direction
- Efficiency: (n-1)!/2 = Θ(n!)

Knapsack Problem

Example: (W=16)

Item	Weight	Value
1	2kg	R200
2	5kg	R300
3	10kg	R500
4	5kg	R100

- Think of yourself as a thief!
- Problem:
  - Given n items, with weights: w1, w2, ... , wn
  - Values: v1, v2, ... , vn a knapsack of capacity W
  - Find the most valuable subset of the items that fit into the knapsack

Knapsack by Exhaustive Search

Subset	Total Wght	Total Value
{1}	2kg	R200
{2}	5kg	R300
{3}	10kg	R500
{4}	5kg	R100
{1,2}	7kg	R500
{1,3}	12kg	R700
{1,4}	7kg	R300
{2,3}	15kg	R800

Subset	Total Wght	Total Value
{2,4}	10kg	R400
{3,4}	15kg	R600
{1,2,3}	17kg	n/a
{1,2,4}	12kg	R600
{1,3,4}	17kg	n/a
{2,3,4}	20kg	n/a
{1,2,3,4}	22kg	n/a

- Enumerate all subsets, calculate total weight and value, choose the best
- Efficiency: $\Omega(2^n)$

Assignment Problem

Example: n = 4

	JOB1	JOB2	JOB3	JOB4
PSN1	9	2	7	8
PSN2	6	4	3	7
PSN3	5	8	1	8
PSN4	7	6	9	4

- Problem:
  - n people and n jobs to be done
  - Each person is assigned to do exactly one job
  - Each job is assigned to exactly one person
  - The cost of person i doing job j is C[i,j]
  - Find a job assignment with the minimum cost

Assignment by Exhaustive Search

	JOB1	JOB2	JOB3	JOB4
PSN1	9	2	7	8
PSN2	6	4	3	7
PSN3	5	8	1	8
PSN4	7	6	9	4

	JOB1	JOB2	JOB3	JOB4	COST
{1,2,3,4}	9	4	1	4	18
{1,2,4,3}	9	4	8	9	28
{1,3,2,4}	9	3	8	4	24
...	...	...	...	...	...

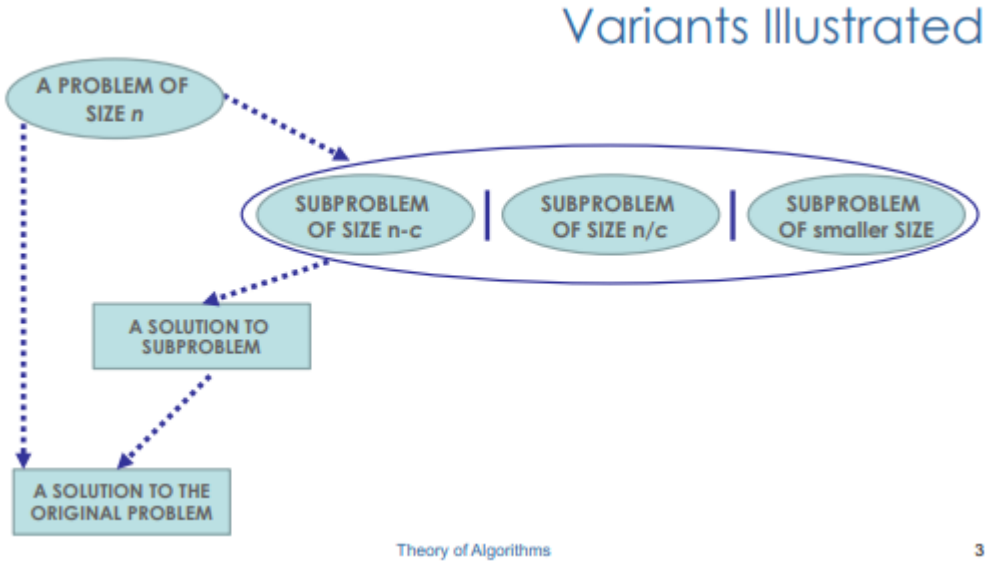
- Select one element in each row in different columns
    - e.g., {1, 4, 3, 2} means PSN 1 gets JOB1, PSN2 gets JOB4, …
    - List all possible assignments
- Solution:
    - Generate all permutations of n positive integers – find the cost of each
    - Computer the total cost for that assignment (by summing the cost of each permutation)
    - Retain the cheapest Poor efficiency: Θ(n!) – we can do better

Comments on Exhaustive Search

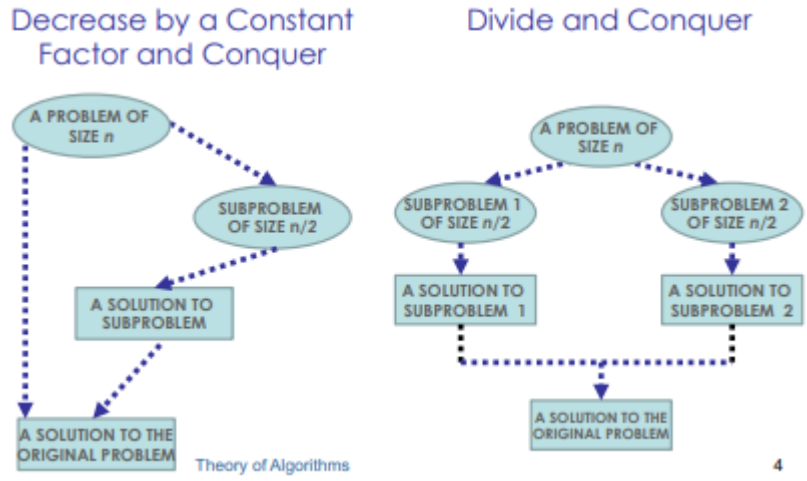
- Exhaustive search algorithms run in a realistic amount of time only on very small instances
- In many cases there are much better alternatives!
- In some cases. exhaustive search (or variation) is the only known solution
- And parallel solutions can speed it up!

Decrease and Conquer

- Strategy:
  - Solve smaller instance – take original problem and shave off some proportion off it
    - Usually solve in a recursive fashion
  - Extend solution of smaller instance to obtain solution to original problem
- Also called inductive or incremental
- 3 Variants:
  - Decrease by a constant
  - Decrease by a constant factor – divide size of problem by a constant
    - Usually, division by 2
  - Variable size decrease – depending on data, might reduce by more/less on each iteration



Divide and Conquer is Different



- Decrease and Conquer throws away half (or constant factor) of the work
    - Binary Search
- Divide and Conquer does both halves – find solution to the halves, and then combine
    - E.g. quicksort

Map of Decrease and Conquer Algorithms

- Decrease by a constant (often 1):
    - Insertion sort
    - Graph Searching: DFS, BFS
    - Generating permutations and subsets
    - Topological sort
- Decrease by a constant factor (usually 2):
    - Binary search
    - Fake-coin problem
    - Multiplication a la Russe
- Variable-size decrease:
    - Euclid's algorithm
    - Interpolation Search
    - Finding the k'th order statistic e.g. the median

Strengths and Weaknesses

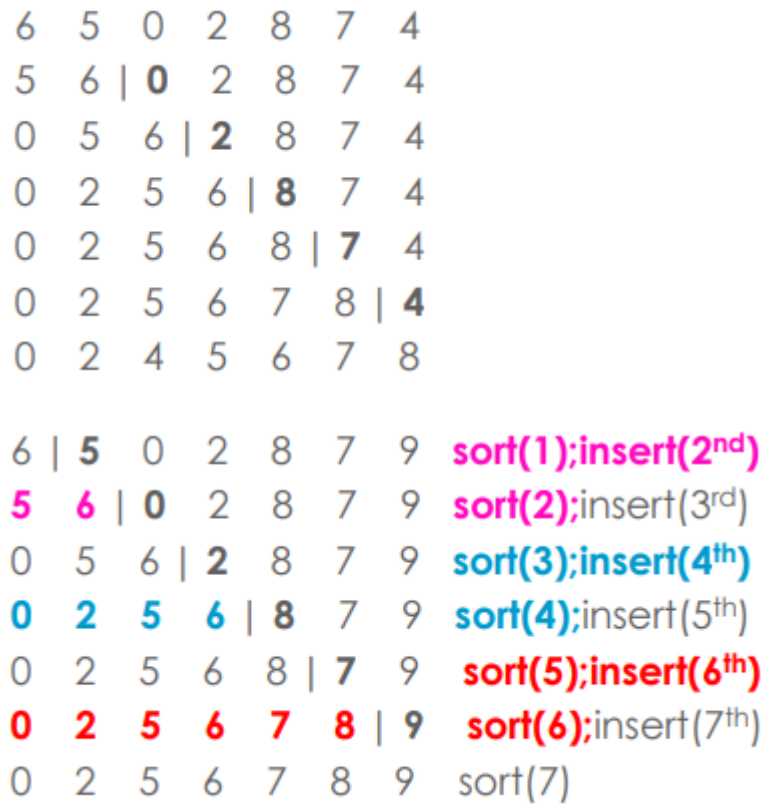
- Strengths:
    - Can be implemented either top down (recursively) or bottom up (without recursion)
    - Often very efficient (possibly Θ(log n) for decrease by constant factor)
    - Leads to a powerful form of graph traversal (Breadth and Depth First Search)
- Weaknesses:
    - Less widely applicable (especially decrease by a constant factor)

Decrease by a Constant

Insertion Sort

- Idea:
    - Take an element
    - Assume an already sorted list of size n-1
    - Insert the remaining element in the correct position – find position(where it should be placed) by searching through the list, of where this element is that has been taken away
    - Before doing this recursively perform insertion sort on smaller list
- Recursive idea but better performed iteratively bottom up
- Decrease by 1 and conquer
- Worst case efficiency – $\Theta(n^2)$, but works well on partially sorted inputs

Example – Bottom-Up Version

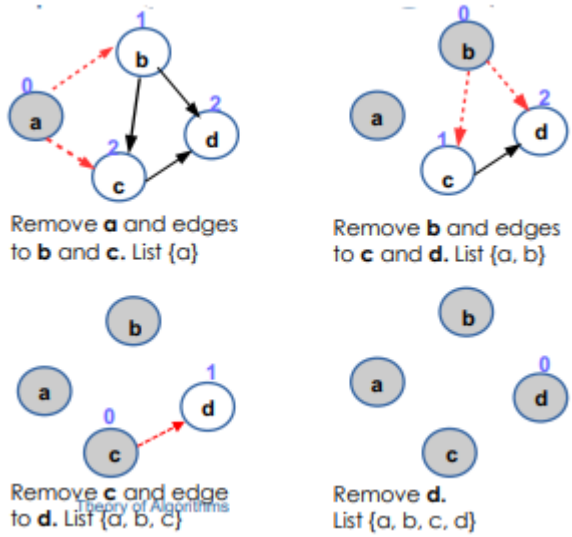


- Take and set aside first element, then insert it into an already sorted list
    - List is of size 1 – inserting 6 into list with just 5 in it
        - 6 goes into 2nd position
    - Now have a list of two elements
- Take 0, and place into already sorted list of [5,6]
- The same for {2,8,7,9}
- Reducing the size of the unsorted portion of the list on each step – until eventually reach a sorted list

Topological Sort

- Problem:
    - In a directed acyclic graph, list the vertices in an order such that edge direction is respected
    - That is for any directed edge the source vertex must appear before the destination in the list
    - Sometimes multiple possible orderings
    - Not solvable if there are cycles
    - Can use Depth-First Search (from last year) to solve
- Applications:
    - Ordering a set of courses that have pre-requisites
    - Evaluating formulae with dependencies in a spreadsheet
    - Ordering tasks in a complex project

Topological Sorting Algorithm

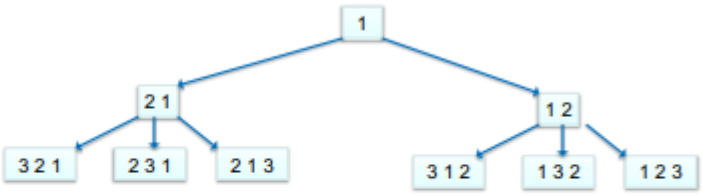


- Solution:
  - Repeatedly remove a source vertex (vertex with only outgoing edges, none coming in) and its incident outgoing edges
  - Append to sorted list
  - If no source vertices exist, either finished or there is a cycle

Generating Permutations

- Necessary as a component of Exhaustive Search
- All possible orderings of numbers in range {1, …, n}
  - Can be used as indices for other objects
- Solution:
  - Generate all (n-1)! permutations of {1, …, n-1}
  - Insert n into each possible position – take a smaller list of numbers (by 1), generate all of those permutations and then, when get back that set of permuations, insert your number into all of the possible positions and that will get you all of the permutations
  - During insert starting from the right or left alternately, satisfies Minimal-Change requirement (next permutation obtained by swapping two elements of previous)
    - This is a useful variation
    - Useful to apply this to algos where this is some cost associated with doing it, and the cost can be updated incrementally
  - Decrease by 1 and conquer

Diagramatically



- Example:
  - Start: 1
  - Insert 2: 12 21
  - Insert 3: 123 132 312 321 231 213

- Reduce problem down to minimal case
- Example: have problem of size 3, with numbers 1, 2, and 3
- Reduce to element 1 – then insert element in the two possible locations
  - Then do the same with the third element
  - With the two halves – right to left with insertions, then left to right

Johnson-Trotter Method

- Alternative permutation generator that avoids permutations of smaller lists – more efficient because it doesn't require you to derive intermediate results (no need for previous tree-like structure)
- Use arrows to keep track of what permutation comes next
  - An element k is mobile if its arrow points to an adjacent element smaller than it
  - E.g. 3 points right, and 2 points left – 2 < 3, thus 3 is mobile but 2 is not

- → ← → ←
  3 2 4 1
- 3 & 4 mobile. 1 & 2 not

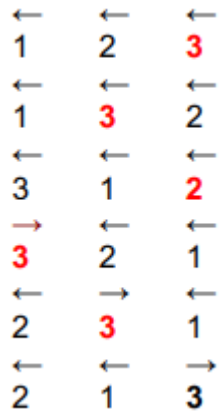
Algorithm

```
Initialize the first permutation with:
    ←   ←       ←
    1   2   …   n
    while last permutation has mobile elem:
         - find its largest mobile element k
         - swap k with neighbour it points to
         - reverse direction of elements > k
         - add the new permutation to the list
    Return the list of permutations
```

- Take initial list of numbers and set each one to be a left mobile number – i.e. attempt to move leftwards
- Then, while you still have mobile elements, repeatedly:
  - Search and find largest mobile element
  - Swop k with its immediate neighbour in the direction in which it's pointing to

    - Reverse the direction of all elements > k – arrows flip direction
    - Print the current state of those lists of numbers as one of the permutations
- Carry on doing this until there is no more mobility, then return the list of permutations

Example



- All elements initialised leftwards mobile
- K = 3 – left-facing and element next to it (in the direction its facing) is less than it
    - Swop 3 with 2 – then reverse the direction of all elements greater than 3 (none)
    - Then print this out as a new permutation – 1 3 2
- 3 is still the largest element – swops with 1, still no larger elements to swop the direction with
    - So just print out 1 3 2, and move the 3
- 3 is now in its leftmost position – can't be moved further left
    - So largest k becomes 2, because 1 < 2
    - Swop 1 and 2
    - Then need to reverse direction of 3's arrow, since 3 > 2
- 3 now becomes largest, most mobile element
    - Moves 1 place to the right
- On next iteration, 3 is still mobile, and is moved
- Now, no longer have any mobile elements – 2 is leftmost, 1 wants to move left but it isn't mobile
    - 3 wants to move right but it's in its rightmost position
- Exit the while loop

Decrease by a Constant Factor

Fake Coin Problem

- Problem:
    - Among n coins, one is fake (and weighs less)
    - We have a balance scale which can compare any two sets
- Algorithm:
    - Divide into two size floor(n/2) piles (keeping a coin aside if n is odd)
    - If they weigh the same, then the extra coin is fake
    - Otherwise proceed recursively with the lighter pile

- Efficiency:
  - |_ _| => floor
  - W(n) = W(|_n/2_|) + 1 for n > 1
  - W(n) = |_$\log_2$ n_| = Θ($\log_2$ n) – applying the Master Theorem
- Can we do better?

Multiplication a la Russe

- Convert multiplication into something that involves summation
- Solution:
  - n*m = (n/2) * (2m) } if n is even
    - ((n-1)/2) * (2m) + m } if n is odd
  - Using doubling and halving instead of multiplication is efficient in hardware (just shift operations!)
  - Decrease by a constant factor of 2 and conquer
- Example:
  - 50 * 20 = (25 * 40) = (12 * 80) + 40 = (6 * 160) + 40
    - = (3 * 320) + 40 = (1 * 640) + 320 + 40 = 1000

Variable Size Decrease

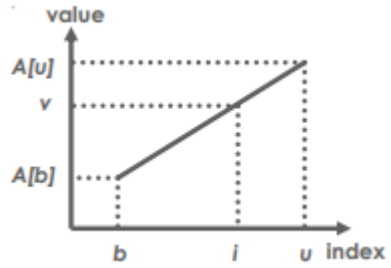
Euclid's GCD

- Problem:
  - Greatest Common Divisor of two integers m and n is the largest integer that divides both exactly
- Euclid's Solution:
  - gcd(m, n) = gcd(n, m mod n)
  - gcd(m, 0) = m
  - Right-side args are smaller by neither a constant size nor factor
- Example:
  - gcd(60, 24) = gcd(24, 12) = gcd(12, 0) = 12

Finding k-th Order Statistic

- Problem:
  - Find the k'th smallest element in a list
  - Median is k = n/2
  - Sorting the list is inefficient
- Solution:
  - Exploit quicksort, noting that pivot is placed in the correct position
  - Partition as usual (≤ pivot | ≥ pivot)
  - Since the pivot ends up in its correct final position, we only need to continue with one of the 2 partitions
  - (if pivot ends up ≥ k'th position, search the first partition, else search the 2nd partition)
- Classified as variable size because quicksort doesn't necessarily partition into equal size sublists
  - Also not divide and conquer because "throw-away" half the list in the process

Interpolation Search

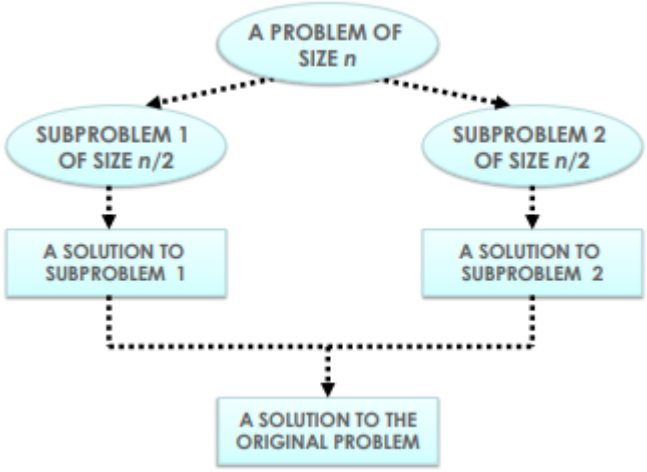


- Mimics the way humans search through a phone book (look near the beginning for 'Brown')
- Know something about the distribution in a list of where things accumulate – can then move to the right part of the search space to look
- Assumes that values between the leftmost (A[b]) and rightmost (A[u]) elements increase linearly
- Expect there to be an even distribution in your list (of what it is you're looking for)
    - Have two values – upper and lower, then predict where the thing is that you're looking for in that list
- Algorithm (key = v, find search index = i):
    - Binary search with floating variable at index i
    - Setup straight line through (b, A[b]) and (u, A[u])
    - Find point P = (x, y) on line at y = v, then i = x
- Diagram:
    - Lower index b, upper index u – have values A[b] and A[u]
    - Looking for value v – find out where v would occur if the values were linearly arranged from A[b] to A[u] (e.g. ¾ of the way between A[b] and A[u])
    - Then adjust search so that it becomes whatever distance away the value is
        - E.g. adjust search so that it is ¾ of the way between then b and u
    - Then check to see if index happens to have that value
        - If not, divide/make the list smaller in the same way as one does with binary search
    - Division is not done evenly

Divide and Conquer

- Have a situation where you halve (and again halve, etc) the problem size
- Best known algorithm design strategy:
  - (1) Divide instance of problem into two or more smaller instances
  - (2) Solve smaller instances recursively
  - (3) Obtain solution to original (larger) inst
- Recurrence Templates apply
  - Recurrences are of the form T(n) = aT(n/b)+ f (n), a >= 1, b >= 2
  - Multiply cost of smaller part (T(n/b)) by a factor of a, and then do some additional work
- Silly Example (Addition):
  - $a_0 + ...+a_{n-1} = (a_0 + ...+a_{n/2-1})+(a_{n/2} + ...a_{n-1})$
  - Adding together a list of numbers – divide that list in half, then recursively add those two halves and combine them with a single addition in the centre

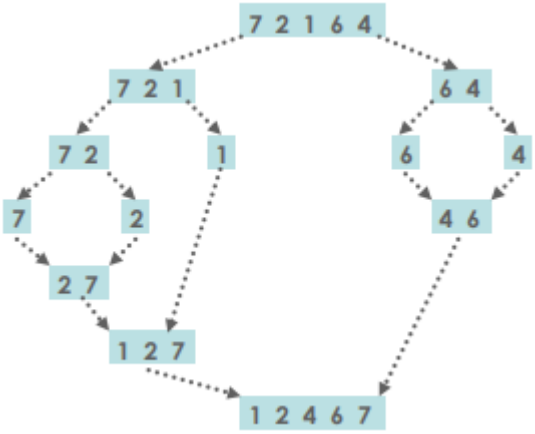
Divide and Conquer Illustrated



MergeSort

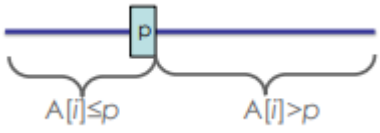
- Efficiency: O(n log n)
- Algorithm:
  - (1) Split A[1..n] in half and put copy of each half into arrays B[1.. n/2] and C[1.. n/2]
  - (2) Recursively MergeSort arrays B and C
  - (3) Merge sorted arrays B and C into array A
- Merging:
  - REPEAT until no elements remain in one of B or C
  - (1) Compare 1st elements in the rest of B and C
  - (2) Copy smaller into A, incrementing index of corresponding array
  - (3) Once all elements in one of B or C are processed, copy the remaining unprocessed elements from the other array into A

MergeSort Example



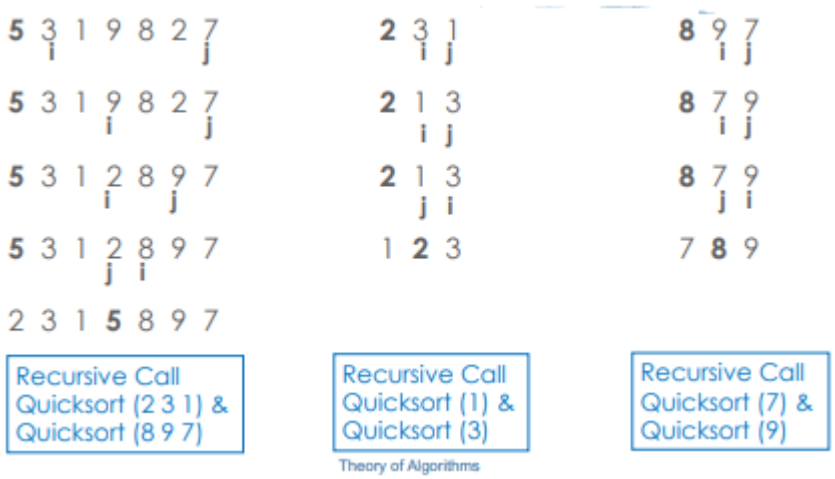
- Master Theorem:
  - $T(n) = aT(n/b) + f(n)$, $f(n) \in \Theta(n^d)$
  - $T(n) \in \Theta(n^d \log n)$ if $a = b^d$
- Recurrence relation:
  - $C(n) = 2\,C(n/2) + C_{merge}(n)$ for $n > 1$, $C(1) = 0$
    - $C(n/2)$ – cost of dividing list, multiplied by 2 (since we're working on both halves)
    - Cost of merging depends on n
  - $C_{merge}(n) = n - 1$ in the worst case
    - Have to do comparisons between the sum of the number of elements in both lists – except for last one, which doesn't require a comparison as ot can just be appended to the end
  - Thus a =2, b =2 and d = 1
- Efficiency (all cases):
  - $a = b^d$ ($2 = 2^1$) in Master Theorem
  - $\Theta(n \log n)$ – for best, worst and avg cases
  - But uses extra space $\Theta(n)$ – need another array of same size as original data
    - E.g. have a million elements/more, could be quite expensive
- Diagram: list [7,2,1,6,4]
  - Split – since list length is odd, won't get even splits
  - Split and split again until get to single elements
  - LHS – 2 < 7, so 2 goes into list, then append 7
    - Then [2,7] and 1 – since 1 < 2, add 1, then append [2,7]
  - Similar for RHS
  - Combine LHS and RHS – put 1 into list, then compare 2 to 4, since 2 <4, put 2 in
    - Compare 7 to 4, since 4 <7, place 4 into list, and so on

QuickSort



- Select a pivot (partitioning element)
- Rearrange the list into two sublists
    - All elements positioned before the pivot are ≤ the pivot
    - Those positioned after the pivot are > the pivot
    - Requires a pivoting algorithm
- Exchange the pivot with the last element in the first sublist
    - The pivot is now in its final position
- QuickSort the two sublists

QuickSort Example



- Choose first element as pivot
    - Note – you wouldn't use this in practice
- Use indices i and j – i starts at the beginning of the remainder of the list and j, which starts at the end
    - Used to swop elements into left and right half
- Increment i whenever the element it's pointing to is <= pivot
- Decrement j whenever the element it's pointing to is > pivot
- Pass 3 and 1 and 7 – when i reaches 9, since 9 > 5 (pivot) and 2 < 5, swop 9 and 2
- Algo continues until j and i meet and swop over – then can stop the partitioning
    - Can insert 5 into the correct position
- Now have two sublists – [2,3,1] and [8,9,7]
    - Independently apply quicksort to these lists

Worst-Case Efficiency of QuickSort

- Intricate – doesn't perform well on sorted list if pivot poorly chosen
- In the worst case all splits are completely skewed
- For instance, an already sorted list!
    - Choose first element as pivot => lop-sided partitioning
    - Have to spend n work to keep the elements to the right of the pivot
- One subarray is empty, other reduced by only one:

- Make n+1 comparisons
- Exchange pivot with itself
- When place pivot, only take away a single element from the list – reducing the size of unsorted portion by 1, not halving as it should be
- Quicksort left = Ø, right = A[1..n-1]
- $C_{worst} = (n+1) + n + ... + 3 = (n+1)(n+2)/2 - 3 = \Theta(n^2)$
  - Because only reducing by 1, and not halving on each step,

General Efficiency of QuickSort

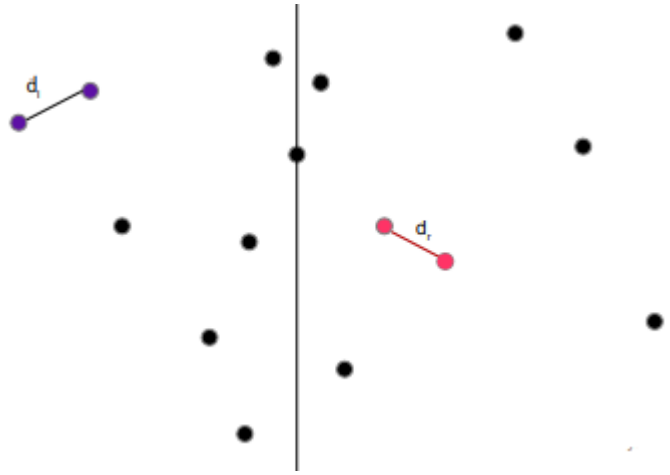
- While worst case is $\Theta(n^2)$, best case (split in the middle) is $\Theta(n \log n)$ and average case (random split) is $\Theta(n \log n)$
- Improvements (in combination 20-25% faster):
  - Better pivot selection: median of three partitioning avoids worst case in sorted files
    - E.g. take one element from start, middle and end of list – take median of those 3
  - Switch to Insertion Sort on small subfiles
  - Elimination of recursion – because recursion has an overhead
- Considered the method of choice for internal sorting for large files ($n \geq 10000$)
  - Favoured over mergesort because it doesn't require additional memory – in place sort

Geometric Problems

Closest Pair by Divide and Conquer

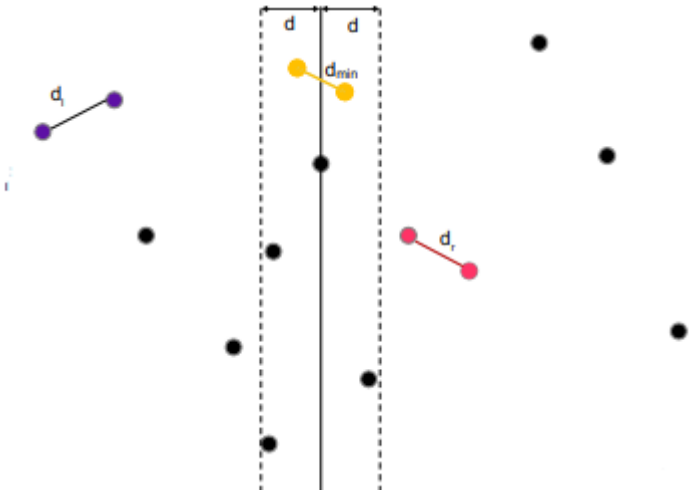
- (1) Sort points according to their x-coordinates – from left to right
- (2) Split the set of points into two equal-sized subsets by a vertical line $x=x_{median}$
- (3) Solve the problem recursively in the left and right subsets, so that we get the leftside and right-side minimum distances dl and d_r. $d_{min}=min(d_l , d_r)$
- (4) Straddle. Find the minimal distance in the set S of points of width 2d around the vertical line. Update d_{min} if necessary
- Divide and conquer because dividing space into two halves i.t.o. the number of points in each half

Closest Pair Illustrated: Split & Solve



- Determine splitting line – solve left and right half
- Recursively find shortest pair in each half
- Let $d_{min} = min(d_l, d_r)$
    - $d_{min} = d_r$ here

Straddle

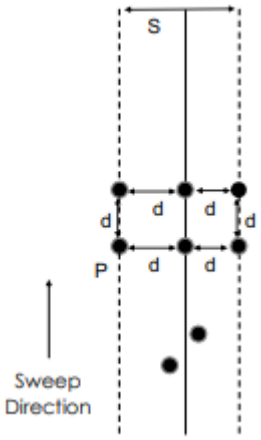


- Search for potential new dmin among points within d of dividing line
- Efficiency:
    - a=2, b=2, d=1
    - $a = b^d$ in Master Theorem
    - Θ(n log n)
    - Same cost as pre-sorting step
    - Algo has two sorts in it – one initial sort from left to right according to x
        - Then within straddle zone, there's a sort vertically in increasing y

Limits in the Straddle Zone

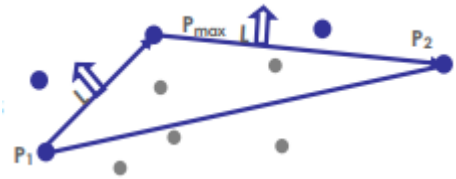
- Initially hard to see how this can differ in efficiency from brute force
    - What if S has almost as many points in it as the non-S part?
    - Then for each point in S we compare it to ALL points in front of it by Brute Force:
    - for i = 0 to n – 2:
        - for j = i + 1 to n-1: etc
    - Efficiency – $Θ(n^2)$
- But there is actually a property of S (straddle zone) that massively limits the number of points we have to look at
    - Limitation depends on sorting points in straddle zone by increasing y

Straddle



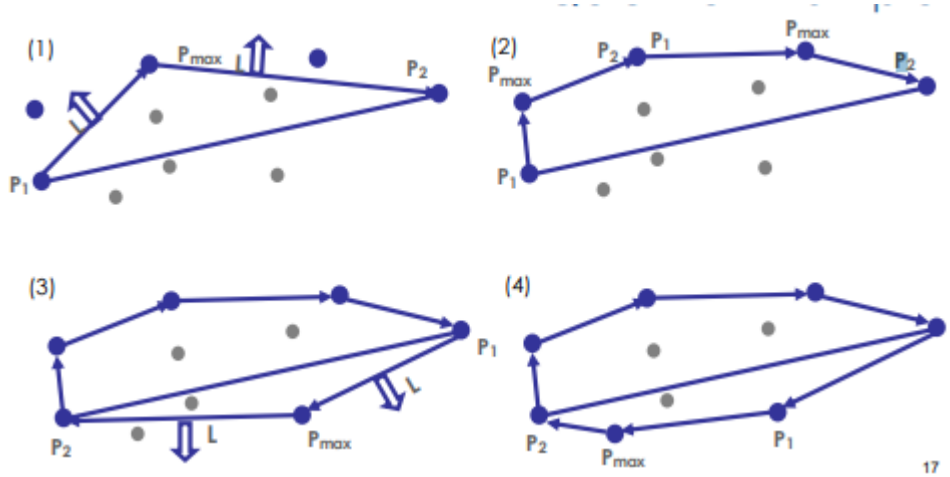
- Assuming point in S are sorted by increasing y.
    - We need only compare against the next 5 points – inner loop thus doesn't iterate over all remaining points, only the next 5
- Proof Sketch:
    - Given the restriction that points to left and right of the dividing line must be at least $d = d_{min}$ apart
    - Configuration on the right represents the closest packing
    - Therefore, not possible for a 6th point to be closer to P than d in the sweep direction

QuickHull



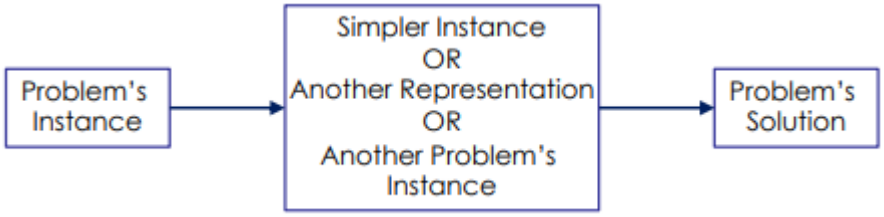
- Solve the Convex Hull problem in an approach reminiscent of QuickSort
- Solution
    - Identify leftmost (minimum x co-ordinate) and rightmost (max x co-ordinate) points P1 and P2
        - "Draw" a line between P1 and P2 – consider points that are to the left (upper hull) and right (lower hull)
        - Division is not necessarily even – can have any number of points in upper, lower hull (as with Quicksort, which doesn't necessarily have an even partition)
    - Compute upper hull:
        - Find point P_{max} that is farthest away from line P1P2
        - Quickhull the points to the left of line $P1P_{max}$
        - Quickhull the points to the left of line $P_{max}P2$
    - Similarly compute lower hull

QuickHull Example



- First step – draw line between leftmost and rightmost co-ordinates
- Second step – find new Pmax for left and right upper hull
- Third step – search for point furthest from line segment P1P2 (notice P1 andP2 have changed positions)
    - Allows one to look always to the left of the line (by swopping endpoints)
    - That then becomes Pmax

Transform and Conquer



- Transform the problem in some way and solve the transformed problem
- Different substrategies:
    - The way in which you transform the problem is important
        - Create simpler example of the same problem – e.g. pre-sort (a array) before trying to solve problem on an array
        - Use another kind of data structure to support solution to the problem – e.g. heaps, balance-search trees
        - Transform instance of current problem into another – then use existing solution of the problem to solve current problem, then transform back
            - Equivalence classes

Approaches

- Instance Simplification = a more convenient instance of the same problem
    - Pre-sorting
    - Gaussian elimination
- Representation Change = a different representation of the same instance
    - Balanced search trees
    - Heaps and heapsort
    - Polynomial evaluation by Horner's rule
    - Binary exponentiation
- Problem Reduction = a different problem altogether
    - Lowest Common Multiple
    - Reductions to graph problem

Instance Simplification

Gaussian Elimination

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ 0 & b_{22} & b_{23} & b_{24} \\ 0 & 0 & b_{33} & b_{34} \\ 0 & 0 & 0 & b_{44} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

- Solution of a fully-determined system of linear equations
- First transform to upper triangular form by Gaussian elimination (simplification) then solve by backward substitution (conquer)

    - i.e. convert initial matric of coefficients into something that is upper diagonal
- [a11…] = coefficients of unknowns
- [x1..] = solution vector
- Convert it into rhs form – once have zeroes lower diagonals => easier to solve
    - Use backwards substitution thereafter

Pre-sorting

- Solve instance of problem by pre-processing the problem to transform it into another simpler/easier instance of the same problem
- Many problems involving lists are easier when list is sorted:
    - Searching
    - Computing the median (selection problem)
    - Finding repeated elements
    - Convex Hull and Closest Pair
- Efficiency:
    - Introduce the overhead of an Θ(n log n) pre-process
    - But the sorted problem often improves by at least one base efficiency class over the unsorted problem (e.g., $\Theta(n^2) \rightarrow \Theta(n)$)
        - If base algo is faster, this pre-sorting slows it down

Is Pre-sorting Better?

- Is transformation by sorting better than brute force?
- Sorting is Θ(n log n) so transformation to sorting is only worthwhile if other algorithms are less efficient
    - Closest pair – $\Theta(n^2) \rightarrow \Theta(n \log n)$, so pre-sort
    - Checking uniqueness – Brute force is $\Theta(n^2)$, so pre-sort
        - Take an element and compare it to every other
    - Finding the min and max – Brute force is Θ(n), so NO pre-sort
        - Since only a single pass of the list is required
    - Finding the median – Brute force is $\Theta(n^2)$, so pre-sort
    - Searching an array – Brute force is Θ(n), so NO pre-sort

Example

- Find the kth smallest element in A[1],..., A[n]
- Special cases:
    - Minimum: k = 1, Maximum: k = n, Median: k = ceiling(n/2)
- Pre-sorting-based algorithm:
    - Sort list
    - Return A[k]
- Partition-based algorithm
    - (Variable Decrease & Conquer)

```
Pivot at A[s] using Partitioning from Quicksort
if s=k return A[s]
else if s<k repeat with sublist A[s+1],..., A[n]
      else if s>k repeat with sublist A[1],..., A[s-1]
```

Notes on the Selection Problem

- Pre-sorting-based algorithm: Θ(n log n) + Θ(1) = Θ(n log n)
    - Θ(1) is constant time fetch of particular element
- Partition-based algorithm (Variable decrease & conquer):
    - Worst case: T(n) =T(n-1) + (n+1) ∈ Θ (n^2)
    - Best case: Θ(n)
    - Average case: T(n) =T(n/2) + (n+1) ∈ Θ (n)
    - Also identifies the k smallest elements (not just the kth)
- Simpler linear (brute force) algorithm is better in the case of max & min

Representation Change

Evaluating Polynomials

- Example:
    - $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$
    - Evaluate for $x = 3$
- The traditional, obvious, brute force approach:
    - $p(3) = 2(3)^4 - 3^3 + 3(3)^2 + 3 - 5$

- Brute Force Polynomial
    - For a polynomial of size n, just the first term $a_n x^n$ – requires n multiplications using brute force
        - This becomes a Θ(n^2) algo
    - We can improve on this by efficiently calculating x^n – i.e. calculate lower order terms and then gradually build up
    - But Horner's rule does even better for large polynomials

Horner's Rule

- Factor x out as much as possible
- Example:
    - $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$
    - $= (2x^3 - x^2 + 3x + 1)\,x - 5$
    - $= ((2x^2 - x + 3)\,x + 1)\,x - 5$
    - $= (((2x - 1)\,x + 3)\,x + 1)\,x - 5$
- So what?

- Can then calculate sub-bracket and fill out nested brackets until get the solution

Horner's Rule vs Brute Force

- Solve: $p(x) = 2x^3 - x^2 - 6x + 5$ for $x = 3$
- By Brute Force:
  - 2.3.3.3-3.3-6.3+5=32
  - 5 mult, 3 add/sub
- By Horner's Rule: $p(x) = ((2x - 1)x - 6)x + 5$
  - C[]: 2 -1 -6 5
  - p: 2 2*3-1=5 5*3-6=9 9*3+5=32
  - 3 mult, 3 add/sub

- With Horner's Rule:
  - Take factorised version and grab coefficients – put that into a coefficient array C[]
  - Then, for given x-value, successively calculate brackets

Horner's Rule Pseudocode

```
double horner(coefficients[0..n], x):
  1 p = coefficients[n]
  2 for i = n - 1 downto 0:
  3     p = x * p + coefficients[i]
  4 return p
```

- Take in array of coefficients and value x
- (1) Set p to first coefficient in the final position
  - Coefficient that responds to innermost nested part of the polynomial
- (2) Iterate for the n-1th polynomial coefficient, down to 0
- (3) Successively multiply x by current running total and adding coefficient for the correct position
- (4) Finally, return evaluated

Horner's Rule Efficiency

- Basic operations are multiplication and addition
  - Let number of multiplications = M(n)
  - Let number of additions = A(n)
  - $M(n) = A(n) = \sum_{i=0}^{n-1} 1 = n$
- For the entire polynomial it takes as many multiplications as the Brute Force method ($\Theta(n^2)$) uses for its first term
- Horner's Rule is extremely efficient

Binary Exponentiation

- Problem:
  - solve a^n using the binary representation of n
- Represent n as a polynomial p(x), with x = 2

- $n = p(x) = b_k x^k + \cdots + b_i x^i + \cdots + b_0$
- Then coefficients of polynomial (b_k…b_j..b_0) are digits of the binary presentation

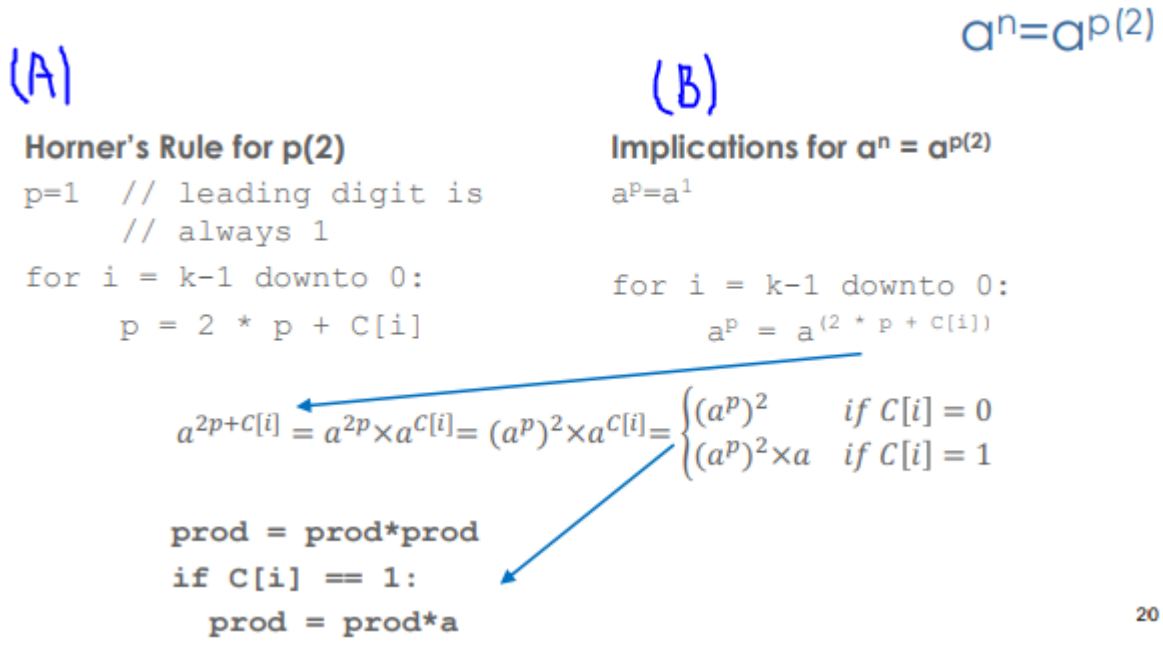
- Example
  - $n = 13, p(x) = 1x^3 + 1x^2 + 0x + 1$
  - Binary 13 is 1 1 0 1
  - Only works if x set to 2

Horner for Exponentiation

- Solve a^n for n = 13 exploiting Horner's Rule
  - $p(x) = 1x^3 + 1x^2 + 0x + 1 = ((x+1)x + 0)x + 1$ at $x = 2$
  - C[]: 1 1 0 1
  - p: 1 1*2+1=3 3*2+0=6 6*2+1=13
  - a^p: a^1 $(a^1)^{2*}a^1$ $(a^3)^{2*}a^0$ $(a^6)^{2*}a^1 = a^{13}$
- Since $a^{n*2+d} = (a^n)^{2*}a^d$ and d is only 0 or 1

- Factorise p(x)
- Calculate p – get 13, as p is a representation of 13
  - Thus, when x = 2, should eval to 13
  - P is just the values of each consecutive bracket (moving inside to out)
- Instead of calculating p, want to calculate a^p – with each step of Horner, take current total, multiple by x and adding coefficient in
  - a3 is the result of the two previous summations
  - If done i.t.o. exponentiation, then they're equivalent
- Didn't need to multiple a by itself 13 times

Algorithm for Exponentiation



- (A) Horner's rule for polynomial, x = 2 and coefficients are either 0/1
    - p=1 – assume highest order coefficient will be 1, because in any binary representation, first digit is never zero
    - Then apply loop for Horner's rules – but with 2 subbed in for x
- (B) Implications
    - First term will always be a^1
    - Then successively build up final a^n by applying relevant formula
    - Can simplify statement
        - Take power term and split into halves – power term that is added == multiplying (with same base)
    - C[i] – coefficient can be either 0 or 1
        - If 0 – then second term falls away
        - If 1 – then it's just a
        - Convert choice to If statement
    - Running total – $(a^p)^2$

General Horner's rule and Horner's rule for Exponentiation

Horner's Rule for p(x)

```
p=C[n]
for i = n-1 downto 0:
     p = x*p + C[i]
return p
```

Horner's Rule for a^n

```
p=a
for i = k-1 downto 0:
     p = p*p
     if C[i] == 1:
          p = p*a
return p
```

Efficiency of Horner Exponentiation

- Basic operation is multiplication, M(n)
- Brute force exponentiation:
    - a^n = a*a…*a (n times)
    - M(n)=n ∈ Θ(n)
- Horner exponentiation:
    - At most 2 multiplications on each loop iteration – assumes every digit is a 1 in binary representation of n
    - k = length of bit representation of n, so k−1 = floor(log2n)
    - k-1 <= M(n) <= 2(k-1)
    - ==> M(n) <= 2 floor(log2n) ∈ Θ(log n)

Problem Reduction

- Used in Complexity Theory to classify problems
- Need to determine if there's an equivalence between 2 different algos/problem classes and transform from one to the other
    - Solve in the other problem space, then covert back
- Computing the Least Common Multiple:

- The LCM of two positive integers m and n is the smallest integer divisible by both m and n
- Problem Reduction: LCM(m, n) = m * n / GCD(m, n) Example: LCM(24, 60) = 1440 / 12 = 120

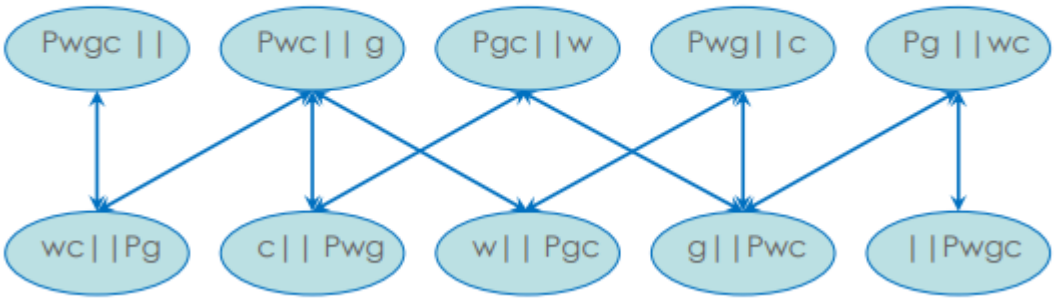
- Reduction of Optimization Problems:
  - Maximization problems seek to find a function's maximum.
    - Conversely, minimization seeks to find the minimum
  - Can reduce between: min f(x) = - max [- f(x)]

Reduction to Graph Problems

- Example:
  - River Crossing Puzzle [(P)easant, (w)olf, (g)oat, (c)abbage]
  - Goat&wolf, and goat&cabbage cannot be left alone
  - Get everyone across the river

- State-Space Graphs:
  - Vertices represent states and edges represent valid transitions between states
  - State represents problem
  - Transition represents a move between one version/state of the problem and another
  - Start (initial state) and goal (solved state) vertices – want to find the shortest number of moves from initial state to the goal
  - Widely used in AI

River Crossing Graph



Strengths and Weaknesses of Transform and Conquer

- Strengths:
  - Allows powerful data structures to be applied
  - Effective in Complexity Theory
- Weaknesses:
  - Can be difficult to derive (especially reduction)

Space and Time Tradeoffs

- Time algo takes to run vs the overall amount of memory it consumes
  - Use more space in order to reduce overall time algo consumes
- Idea: Trading off space for time
  - Use more memory to reduce computation cost
  - Good idea if need to do the same task multiple times using some subset of the inputs (to start with)
- Input enhancement (preprocessing)
  - Store ancillary information about the problem instance which will make subsequently solving it faster
  - Take problem instance, do preprocessing on it, build up a table that will help accelerate the problem solving stage later on
  - Examples: sorting by counting, Horspool and Boyer-Moore's string matching
- Prestucturing
  - An initial step to make access to the data faster
  - Examples: hashing, indexing with B-trees
- Dynamic programming fits into this category but will be considered separately

Sorting by Counting

Idea

- Sort a list whose elements fall in a restricted range of integers [L … U]
  - Possible to use pre-processing to make actual stage of sorting it linear
    - Whilst sorting algos are typically Θ(n log n) in the best case, this applies for a general sort that is unrestricted
  - Here, have a range within some lower and upper integer
- Using a frequency table that counts the number of occurrences of each element
  - With U-L entries in the table
- And a distribution table derived from frequencies to tell us where to place elements
  - Convert frequency table to distribution table
  - "What is the start of the repeated numbets?"
    - E.g. if 56 is repeated 3 times, where do we put it in the list?

Sorting by Counting Example

Sorting by Counting Example

Example: A =

13	11	12	13	12	12

Array Value	11	12	13
Frequencies	1	3	2
Distribution	1	4	6

	D[0]	D[1]	D[2]	S[0]	S[1]	S[2]	S[3]	S[4]	S[5]
A[5] = 12	1	4	6				12		
A[4] = 12	1	3	6			12			
A[3] = 13	1	2	6						13
A[2] = 12	1	2	5		12				
A[1] = 11	1	1	5	11					
A[0] = 13	0	1	5					13	

Efficiency: Θ(*n*) - Best so far, but highly restricted inputs

4

- Have list A
- Frequency table has a low of 11 and an upper of 13 – 3 elements
    - Run through list once in Θ(n) time, calculating the number of occurrences of the integers
    - Then do a prefix sum – 3+1 =4, 4+2 = 6
        - Shows what position in the output, those numbers will appear in
- Do final run-through, grab elements into input array in linear time, and place into their final position
    - Start from last element in the array and work backwards
- Start with 12 – maps into middle element (using distributional array)
    - Maps to 4 – gets placed in index 3 in the array
    - Once this has been done, subtract 1 from 4 – so any other 12s will be position before the first 12 in the array
- Similar for A[4] to A[0]
- One linear iteration to get frequencies, another to get through frequency list to calculate distribution positions
- Then final iteration through list to sort it
- Thus, have Θ(n) efficiency – but only with a restricted subset

Sorting by Counting Algorithm

```
for j ← 0 to U-L do D[j] ← 0                    // init freq
for j ← 0 to n-1 do D[A[i]-L] ← D[A[i]-L]+1    // calc freq
for j ← 0 to U-L do D[j] ← D[j-1]+D[j]         // calc distrib
for i ← n-1 downto 0 do
     j ← A[i] - L
     S[D[j]-1] ← A[i]
     D[j] ← D[j]-1
return S
```

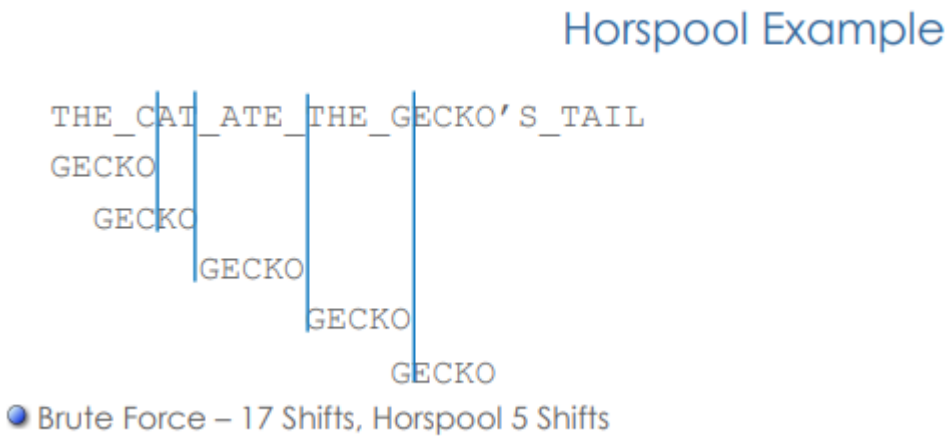
- Constraints:
- Elements in [L…U], list A with n elements to be sorted
- Move original elements into correct positions in a new list
- Allows additional data to be attached to each element

Horspool's String Matching

Idea

- Start matching from the end of the pattern and on a mismatch shift the pattern by more than a single space
- Use a shift table T:
    - One entry per character c
    - Gives the no. of places to shift the pattern when text character c is aligned with (opposite) the last character of the pattern
    - Maximum shift is m (the length of the pattern)

Horspool Example

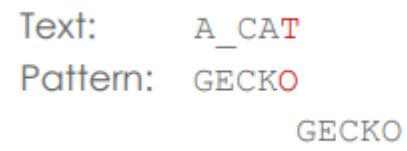


Horspool's Algorithm

- Align the left-most character of the pattern to the left-most character of the text (as in brute force)
- Match pattern against corresponding text from right to left (i.e. backwards through the pattern)

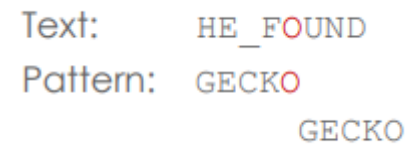
- If our pattern mismatches there are four cases to consider
- In all cases compare the last letter of the pattern against the last corresponding letter in the text

Case 1



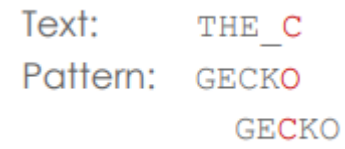
- If the last letter is not in the pattern: (T not in GECKO) Move the entire pattern past the last letter
    - i.e., shift by m

Case 2



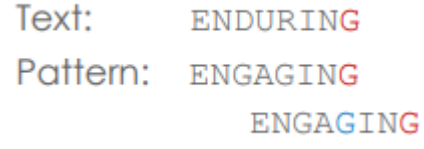
- If last letters match but that letter isn't in rest of pattern, move entire pattern past the last letter
    - (exactly as in case 1)
    - i.e., shift by m

Case 3



- If last letter is in the pattern (C is in GECKO) but not at the end
    - Align with rightmost occurrence of that letter in the pattern
    - i.e., T(c) = m - 1- pos(c) in pattern
        - m = 5, m – 1 – 2 (index 2)
        - Thus, shift = 2

Case 4



- If last letters match and that letter is elsewhere in the pattern
- Align with rightmost occurrence of that letter in pattern, ignoring the last character

- (exactly as in case 3)

Horspool Summarised

- 1. Construct a shift table T
- 2. Align the pattern against the beginning of the text
- 3. Repeat until match or pattern reaches end text:
    - Starting with the last character of the pattern compare the corresponding characters in the text until either all m characters matched; or mismatch found
    - If match, report index position of first position of pattern in the text
    - On mismatch retrieve T(c) where c is character in text aligned to last character in pattern.
    - Shift pattern right T(c) positions.

Example Shift Table

- Example: ENGAGING
    - m = 8
    - There are 5 unique letters

T(E) = m – 1 – rightmost index of E = 8 – 1 – 0 = 7
T(N) = 8 – 1 – 6 = 1
T(I) = 8 – 1 – 5 = 2
T(G) = 8 – 1 – 4 = 3
T(A) = 8 – 1 – 3 = 4
T(All others) = m = 8

E	N	G	A	I
7	1	3	4	2

- Pattern = ENGAGING
    - Only 5 unique characters
- Build up character table with all ASCII characters in it
    - All initialised to 8 – for ones that appear in pattern, calculate how much to shift by
- For ones that appear in pattern, calculate how much to shift by
    - E – find rightmost index that isn't at the very end (but ignore last character)
        - 8 – 1 – 0
    - Similar for N, I, G, A
    - All others have m = 8

Example 1

```
Text:     IT_WAS_ENGAGING
Pattern: ENGAGING
                 ENGAGING
```

- Mismatch, last letter = E
- T(E) = 7, so shift pattern 7 places

E	N	G	A	I
7	1	3	4	2

Theory of Algorithms

- Pattern appears in text, but not in initial position
- Run check from right to left – fail on first match of pattern against text
- Instead, match G from pattern to E in test
    - Lookup in shift table – see the shift for E is 7
- Thus, shift pattern 7 places to the right, and then it matches

Example 2

```
Text:     IT_WAS_VERY_ENGAGING
Pattern: ENGAGING
                 ENGAGING
```

- Mismatch, last letter = V
- T(V) = 8, for all character not in the first 8-1 positions of the pattern

E	N	G	A	I
7	1	3	4	2

- Align on the left side – perform check
    - Fail with V against G
- Lookup in shift table – V is not in pattern, so gets default shift of max, since m =8, shift 8 to the right
- It would run and mismatch until final shift

Example 3

```
Text:    IT_WAS_NOT_ENGAGING
Pattern: ENGAGING
          ENGAGING
```

- Mismatch, last letter = N
- T(N) = 1, so shift pattern 1 place

E	N	G	A	I
7	1	3	4	2

- G fails against N – go to lookup table
    - But now, only shift by one place

Example 4

```
Text:    IT_WAS_GOING_ENGAGINGLY
Pattern: ENGAGING
             ENGAGING
```

- Mismatch, last letter = G
- T(G) = 3, so shift pattern 3 places

E	N	G	A	I
7	1	3	4	2

- G matches with G
- Then move and check N with underscore – fails
- Check in shift table for G – shift by 3

Boyer-Moore

Idea

- Based on same two ideas as Horspool:
    - Compare pattern characters to text from right to left
    - Given a pattern, create a shift table that determines how much to shift the pattern
- Except:
    - In addition to bad-symbol shift table
    - Uses a good-suffix shift table with same idea applied to the number of matched characters
- Efficiency:
    - Horspool's approach is simpler and at least as efficient in the average case

Bad Symbol Shift

- Example:
  - Horspool:
    ```
    ???SER
    BARBER
       BARBER
    ```
  - Boyer-Moore:
    ```
    ???SER
    BARBER
        BARBER
    ```
  - Since the S is not in BARBER move the pattern past the S

- Rule for calculating shift is called the bad character rule
  - It needs a shift table (use Horspool as the basis)
  - Look up the shift of the rightmost mismatched char (not at the end as in Horspool)
- In example, Boyer-Moore will throw mismatch and then shift on the S, not R (as Horspool does)

Bad Symbol Shift Rule

- Let m = length of string
- Let T = shift table as in Horspool
- Let k = number of matched chars from back
- Let c = mismatched char in text
  - bad_shift = max(T(c) – k , 1)
  - Note T(c) – k could be less than 1, hence the max operation.
    - We always can shift at least 1 position as in brute force

Difference in Shifts

```
TANSER
BARBER
    BARBER // Boyer-Moore: max(T(c)-k, 1) = max(6-2, 1) = 4
   BARBER  // Horspool: T(R)= 3
// Here Boyer-Moore beats Horspool
MY AEROPLANE
BARBER
  BARBER  // Boyer-Moore: max(T(c)-k, 1) = max(4-2, 1) = 2
   BARBER // Horspool: T(R)= 3
// Here Horspool beats Boyer-Moore
```

- Barber is pattern, and Tanser is text
  - Boyer-Moore – try and match, succeed and R, E and fail on S
    - Then lookup value in the shift table for S – since S not in pattern, shift is max = 6
    - Check how many chars were successfully matched (2) – then say 6 – 2
    - Take the max(4,1) – thus shift is 4
  - Horspool – look at last char R and what it's shift value is
    - Shift is thus 3
  - Boyer-Moore has greater shift and wins here
- My aeroplane is text, barber is pattern

- Boyer-Moore – succeed on R,E and fail on A
  - Lookup value of A – 4 shifts, subtract 2 (# of successful matches)
  - Thus shift 2
- Horspool – lookup R, shift is 3
- Here, Horspool beats Boyer-Moore

Pre-processing

- Before we start searching, we know everything about the pattern and nothing about the text
  - The shift table is set up ONLY using information from the pattern
- Both algorithms try to extract useful information from the pattern in advance of the search, in order to maximise the size of the shift they do on each mismatch
- This is the point: The shift table is determined solely by the properties of the pattern, not the text
- It helps to understand this when we consider the next part of Boyer-Moore: Good Suffix Rule

Good Suffix Rule

- This is the second shift rule calculated by Boyer Moore
  - Boyer Moore selects whichever shift is greater of the good suffix versus bad symbol shift rule
  - Based on number of characters that were successfully matched before had first failure
- Several Cases:
  - Case 1 – The matching suffix does not occur elsewhere in the pattern
  - Case 2 – The matching suffix occurs somewhere else in the pattern
  - Case 3 – A part of the matching suffix occurs at the beginning of the pattern
- Build a shift table G(k) that depends on number of matches
  - Parameter k is # of previously successful matches

Case 1

- Matching suffix does not occur elsewhere in the pattern
- Can shift pattern past the text
- Example:
  - Text: . . A B A B . . .
  - Pattern: M A O B A B
  - Successfully match BAB but fail on MAO
  - Determine G(k = 3) (k = the number of matched chars)
  - Consider the suffix BAB, which is G(3), nowhere else in the pattern is there a BAB
  - So we can shift the entire pattern, G(3) = m = 6

Case 2

- The matching suffix occurs elsewhere in the pattern
- Case 2A: Text: . . . . A B A B . . .
  - Pattern: O B A B O B A B
  - Shift: O B A B O B A B
  - Have suffix and character that fails before that suffix also occurs before the pattern of the suffix
  - Same mismatch letter recurs in pattern so full shift, G(k=3) = 5
- Case 2B:

- Text: . . . . A B A B . . .
- Pattern: A B A B O B A B
- Shift: A B A B O B A B
- Character of mismatch doesn't occur before the good suffix
- Mismatch does not recur so cannot do a full shift, G(k=3) = 4

Case 3

- A part of the matching suffix occurs at the beginning of the pattern
- Example:
  - Text: . . . . . A B A B . . .
  - Pattern: A B C B A B
  - Shift: A B C B A B
  - Here the front letters match part of the suffix
  - So we can only shift G(k=3) = 4

Boyer-Moore Algorithm

- Good-suffix shift:
  - If k symbols matched before failing, shift the pattern up G(k) positions
- Bad-symbol shift:
  - If k symbols matched before failing, shift the pattern up max(T(c) – k, 1) positions (where c is the character that didn't match)
- Algorithm:
  - 1. Build tables T and G
  - 2. When searching, use either bad-symbol shift or good-suffix shift, whichever is larger

Summary

- Preprocess all or part of the problem instance to create an acceleration data structure
- Can be very effective if the acceleration structure is used repeatedly in the subsequent algorithm
- But must consider the cost of preprocessing in the overall algorithm efficiency

Dynamic Programming

Principle

- Solve problems by storing optimal solutions to overlapping subproblems
  - Each subproblem solved only once and stored for lookup
  - Thus, falls into the class of space-time tradeoffs
- Solution derived from a recurrence relation specification
- Must obey principle of optimality – i.e. optimal subproblems must themselves represent the optimal solution
  - An overall optimal solution can be derived from optimal solutions to sub-instances

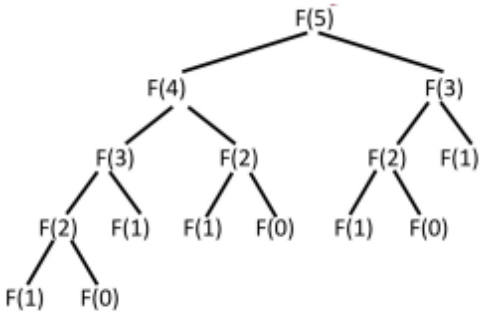
Dynamic Programming Algorithms

- Introductory Algorithms
  - Fibonacci
  - Change-making
- Transitive Closure: Warshall's Algorithm
  - Basically, reachability – can one vertex in a graph reach the other?
- All Pairs Shortest Path: Floyd's Algorithm
- Knapsack Problem

Fibonacci

- F(n) = F(n-1)+F(n-2)
- Brute Force

```
fibo(n):
      if (n == 0) return 0
      if (n == 1) return 1
      return fibo(n-1) + fibo(n-2)
```



- Dynamic Programming

```
fibo(n):
      F[0] = 0
      F[1] = 1
      for i = 2 to n do:
            F[i] = F[i - 1] + F[i - 2]
      return F[n]
```

F[0]	F[1]	F[2]	F[3]	F[4]	F[5]
0	1	1	2	3	5

5

Change Making Problem

- Given a collection of coin denominations d1 < d2 < … < dm
  - Find exact change for the amount n such that the minimum number of coins is used
  - Each denomination has an unlimited number
  - d1 = 1

- For typical sub-cases there is an efficient greedy algorithm
    - Here we consider the general case

Change Making Algorithm

- Recurrence Relation
    - Add one coin to the smallest number of coins for different denominations subtracted from the current total

      $F(n)$ is minimum number of coins adding up to n

      $F(0) = 0$

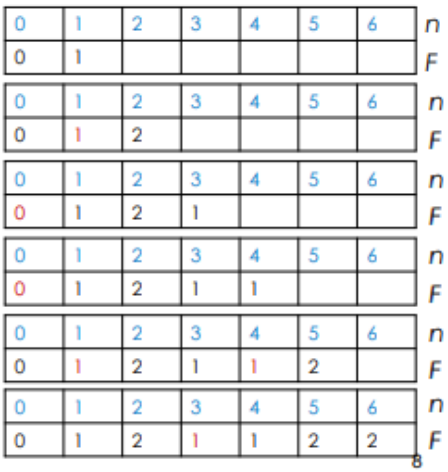
      $F(n) = \min_{j:n \geq d_j}\{F(n) - d_j\} + 1 \ \ for\ n > 0$
    - F(n) tells the number of coins that have been used
        - So, for the current value, iterate of j, where j is all indices available in denomination array – so long as that denomination isn't greater than current the current n
            - i.e. can't use a coin that's more valuable than what trying to get change for
            - Take the min of that
            - Because using an extra coin of that denomination, add 1 to relation
        - Denomination array – stores different denominations that can be used
- Algorithm:
    - Iterate from 1 to n calculating F

```
F[0] ← 0
for i ← 1 to n do
    temp←∞; j←1
    while j ≤ m and i ≥ D[j] do
        temp ← min(F[i - D[j]], temp)
        j←j+1
    F[i] ← temp + 1
return F [n]
```

    - *temp* stores the current max
    - Set j (iterates over denominations) to 1 and carry on iterating through j – so long as haven't looked through all denominations or haven't reached the stage where the current denomination is greater than the value we're looking for
    - For each of those denominations, check to see that it's less than current temp value
        - Current value – denomination vs temp – update temp
    - Increment j, then check next denomination
    - Will get number of coins required to obtain that minimum value – but using an extra coin, so need to update the current one by adding 1

Change Making Example

- n=6, D = [1,3,4], F[0] = 0
  - i=1: F[1]=min{F[1-1]}+1=1
  - i=2: F[2]=min{F[2-1]}+1=2
  - i=3: F[3]=min{F[3-1],F[3-3]}+1=1
  - i=4: F[4]=min{F[4-1],F[4-3],F[4-4]}+1=1
  - i=5: F[5]=min{F[5-1],F[5-3],F[5-4]}+1=2
  - i=6: F[6]=min{F[6-1],F[6-3],F[6-4]}+1=2

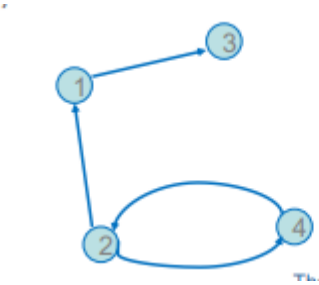


- Assume have R6 that have to get change for
- Denominations are R1, R3 and R4
- i=1 => What is the minimum amount of change needed to get R1 value?
  - Iterating over 1 – (all denominations that're valid)
  - Here, have single denomination that is valid – R1
    - Shouldn't go for higher denomination – because then F[1-3] produces negatives (not allowed)
  - 1 gets put in corresponding position in table
    - Entry in table basically means – for R1, what is the minimum number that we need to get change for that
    - It so happens that R1 is a coin of value 1
- i=2 => Look backwards in table and look at denominations available, assuming its valid
  - Only valid denomination is again R1 – so only 1
  - Thus have F[2-1] – which is F[1], which is 1
  - Add 1, gives 2
- Skipping ahead to i=5:
  - Can use all denominations since is 5 sufficiently large to prevent negatives (when denominations are subtracted)
    - {F[5-1],F[5-3] and F[5-4]} => F[4], F[2] and F[1]
    - Find min of these – F[4] or F[1] and add 1 = 2

Transitive Closure

- For a directed graph with n vertices
- Transitive closure is the n-by-n Boolean matrix T = $\{t_{ij}\}$ indicating reachability
  - If there is a 1 in the ith row and jth column then there is a directed path from vertex i to vertex j
  - If the entry is 0 then there is no path

Example:



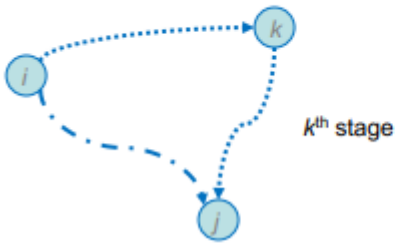
$$T = \begin{array}{c} \\ \text{from} \end{array} \begin{array}{c} \text{to} \\ \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix} \end{array}$$

Warshall's Algorithm Principle

- Idea: if there's a way to go from A to B, and also from B to C, then there's a way to get from A to C
- Key idea is to build up partial solutions – build an adjacency matrix
  - Adjacency matrix – if there is a directed edge between two vertices, then they're directly reachable
  - Then check each vertex in turn and check if it can be added onto the paths
- A path exists between two vertices i, j, iff there is
  - an edge from i to j; or
  - a path from i to j going through vertex 1; or
  - a path from i to j going through vertex 1 and/or 2; or
  - a path from i to j going through vertex 1, 2, and/or 3; or
  - …
  - a path from i to j going through any of the other vertices
- Successively build corresponding transition matrices $R^{(0)}$, $R^{(1)}$, …, $R^{(n)} = T$

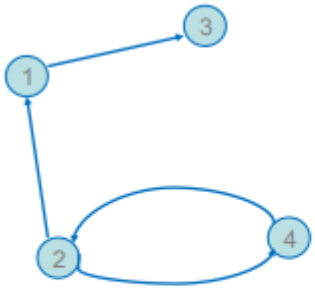
Warshall's Algorithm

$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path only via } 1,\ldots,k-1) \\ \textbf{or} & \\ R^{(k-1)}[i,k] \textbf{ and } R^{(k-1)}[k,j] & \begin{pmatrix} \text{path from } i \text{ to } k \\ \text{and from } k \text{ to } i \\ \text{using just } 1,\ldots,k-1 \end{pmatrix} \end{cases}$$



- In the kth stage determine if a path exists between two vertices i, j using just vertices among 1,…,k
- Recurrence relation – kth stage, trying to include vertex k in our paths
  - Either there is already a directed path that happens to not include k but includes i (from i to k-1) – in this case, no change needed since Boolean value is already there and valid
  - Otherwise, check if there's a path from vertex i to k and vertex k to j – then join those two paths together, which creates a valid path

Warshall's Algorithm Example: R^0

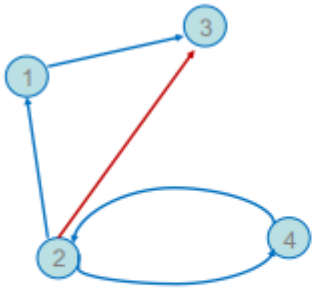


$$R^0 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

- Note R^0 is the Adjacency Matrix
  - The *n*-by-*n* Boolean matrix where a 1 in row i column j indicates a directed edge from i to j

- Adjacency matrix – representation of graph showing directed edges between vertices
  - 1 represents such an edge
-

Warshall's Algorithm Example: R^1



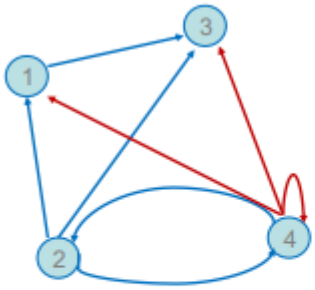
$$R^0 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

$$R^1 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

- Change Boolean from 0 to 1 at (i, j)where column j in row 1 and row i in column 1 both have 1's

- Now examine matrix – find intersections between columns and row that both have 1s in them
  - Can thus show that there is a route that incorporates that vertex
- Example shows a route from vertex 2, to vertex 3, that incorporates vertex 1 – [2,1,3]
  - Vertex 1 is incorporated into transitive closure

Warshall's Algorithm Example: R^2



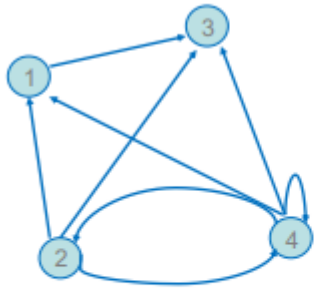
$$R^1 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

$$R^2 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \end{matrix}$$

- Change Boolean from 0 to 1 at (i, j)where column j in row 2 and row i in column 2 both have 1's

- Consider row 2, column 2 and follow same procedure as mentioned
    - As seen in diagram – fill in 1s for routes that start at vertex 4
- Note: on the diagonal – don't include vertices going back to themselves, unless there is a path that does that
    - In the example, the path is [4,2,4] – hence the 1 entry on the rightmost column of the bottom row of the matrix

Warshall's Algorithm Example: R^3



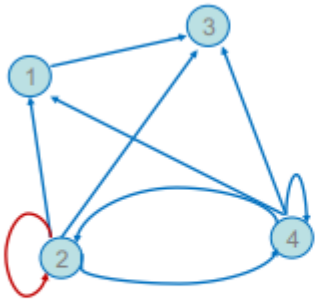
$$R^2 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

$$R^3 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

- Change Boolean from 0 to 1 at (i, j)where column j in row 3 and row i in column 3 both have 1's
    - No new paths because vertex 3 is a sink

- Column 3, row 3 – row is all 0s because vertex 3 is a sink
    - i.e. No outgoing edges from vertex 3

Warshall's Algorithm Example: R^4



$$R^3 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

$$T = R^4 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

- Change Boolean from 0 to 1 at (i, j)where column j in row 4 and row i in column 4 both have 1's

- Row 4, column 4 – again, update new routes
  - This route actually ends up taking 2 back to itself – [2,4,2]
- Transitive closure matrix is now complete

Warshall's Algorithm Code

```
1 R(0) ← adjacency matrix
2 for k ← 1 to n do
3     for  i ← 1 to n do
4         for  j ← 1 to n do
5             R(k)[i,j] ← R(k-1)[i,j] or (R(k-1)[i,k] and R(k-1)[k,j])
6 return R(n)
```

- (1) Set R^0 to adjacency matrix
- (2) Consider all other vertices i.e. Rk, k equals {1,…,n}
- (3) and (4) – for each iteration, go over every single vertex in the matrix, hence the nested for loops
- (5) Using boolean *ands* and *ors*
  - For each of those vertices, either – select an existing route OR consider an and of the route that goes from i to k, and from k to j
- (6) Return R^n as transitive closure
- Complexity:
  - Time efficiency in $O(n^3)$
  - Space efficiency in $O(n^3)$ - Possible $O(n^2)$

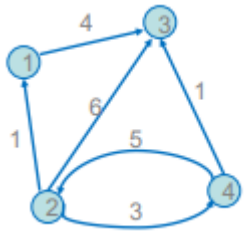
All Pairs Shortest Path

- Tabulate distances between pairs of points on the map
  - For scheduling airlines/trains, etc.
- Cast in graph terms
  - For any two vertices, tells us the shortest path between them
- Floyd's Algorithm

- Same idea as Warshall's Algorithm
- Create matrix , but stores distances instead of booleans

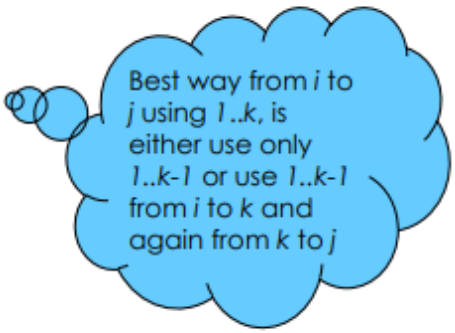
Floyd's Algorithm

- In a weighted graph (directed or undirected), find shortest paths between every pair of vertices
- Construct solution using series of matrices $D^{(0)}$, $D^{(1)}$, …, $D^{(n)}$
  - Using an initial subset of the vertices as intermediaries
  - Same idea as Warshall
- Case study graph:



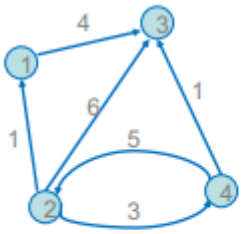
Floyd's Idea

- Initialisation:
  - $d_{ij}^{(0)} = w_{ij}$ direct
  - This is the weighted adjacency graph
  - No intermediate vertex from i to j
- Iteration:
  - $d_{ij}^{(k)} = \mathbf{min}(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ if $k \geq 1$
- Termination:
  - Stop at k = n
  - All intermediate vertices are in set {1, 2, …, n}



- Start with matrix D0 – if there is a direct edge between two vertices, then put weight of edge into matrix
  - No need for intermediate edges between 2 vertices
- Then iterate
  - Have a recurrence relation
  - For current matrix k, use previous matrix – achieve by using min(weight of previous entry, include vertex k in route – cost to vertex k from i + from vertex k to j)
- Stop once k = n, because all intermediate possible vertices have been included

Floyd's Algorithm Example: $D^{(0)}$



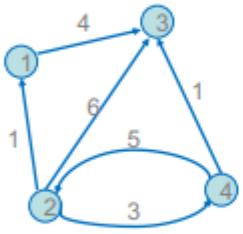
$$D^0 = W = \begin{matrix} 0 & \infty & 4 & \infty \\ 1 & 0 & 6 & 3 \\ \infty & \infty & 0 & \infty \\ \infty & 5 & 1 & 0 \end{matrix}$$

- Initialise with the weighted adjacency matrix. Place infinities where there is no directed edge

- ∞ When there isn't a directed edge between two vertices, otherwise include weight of edge
- 0 entries on diagonal – moving from vertex 1 to vertex 1 => don't have to move at all

Floyd's Algorithm Example: $D^{(1)}$

Floyd's Algorithm Example: $D^{(1)}$



$$D^0 = W = \begin{matrix} 0 & \infty & 4 & \infty \\ 1 & 0 & 6 & 3 \\ \infty & \infty & 0 & \infty \\ \infty & 5 & 1 & 0 \end{matrix}$$

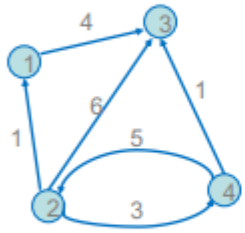
$$D^1 = \begin{matrix} 0 & \infty & 4 & \infty \\ 1 & 0 & 5 & 3 \\ \infty & \infty & 0 & \infty \\ \infty & 5 & 1 & 0 \end{matrix}$$

- At (i, j) take min of column j in row 1 + row i in column 1 and the current element

- To go to D1 – either use existing cost in previous matrix OR, intersect column 1, row 1
  - Add 4 + 1 = 5, 5 < 6 (current value), thus use new shortest route 5
  - Includes vertex 1
- Thus route from 2 to 3 must go through vertex 1

Floyd's Algorithm Example: D(2)

Floyd's Algorithm Example: D(2)



$$D^1 = \begin{matrix} 0 & \infty & 4 & \infty \\ 1 & 0 & 5 & 3 \\ \infty & \infty & 0 & \infty \\ \infty & 5 & 1 & 0 \end{matrix}$$

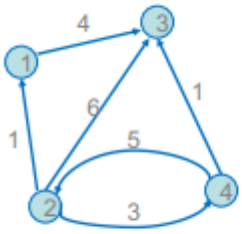
$$D^2 = \begin{matrix} 0 & \infty & 4 & \infty \\ 1 & 0 & 5 & 3 \\ \infty & \infty & 0 & \infty \\ 6 & 5 & 1 & 0 \end{matrix}$$

- At (i, j) take min of column j in row 2 + row i in column 2 and the current element

  - Take row 2 and column 2 – check if sum of their values is less than current one
  - Thus, take 5 + 1 = 6, 6 < ∞
  - Thus, have new route from vertex 4 to 1 through vertex 2
  - Remember, using D2 implies that trying to *include* vertex 2 in our routing

Floyd's Algorithm Example: D(3)

Floyd's Algorithm Example: D(3)



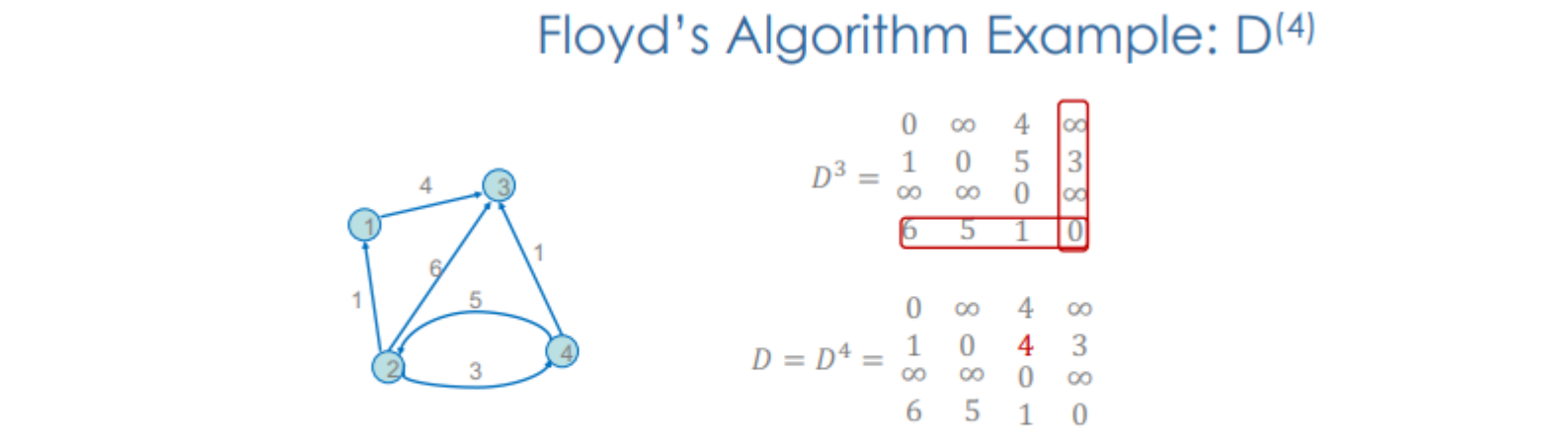
$$D^2 = \begin{matrix} 0 & \infty & 4 & \infty \\ 1 & 0 & 5 & 3 \\ \infty & \infty & 0 & \infty \\ 6 & 5 & 1 & 0 \end{matrix}$$

$$D^3 = \begin{matrix} 0 & \infty & 4 & \infty \\ 1 & 0 & 5 & 3 \\ \infty & \infty & 0 & \infty \\ 6 & 5 & 1 & 0 \end{matrix}$$

- At (i, j) take min of column j in row 3 + row i in column 3 and the current element

  - Vertex 3 is a sink – so none of the intersections will change anything

Floyd's Algorithm Example: $D^{(4)}$



- At (i, j) take min of column j in row 4 + row i in column 4 and the current element

- Row 4, column 4 – 3+1 = 4, 4 < 5 so update
  - Have new route that goes through vertex 4
- Now have all pairs shortest path matrix (D)

Floyd's Algorithm Code

```
1 D ← W
2 for k ← 1 to n do
3    for i  ← 1 to n do
4       for j  ← 1 to n do
5          D[i,j] ← min( D[i,j],  D[i,k] + D[k,j] )
6 return D
```

- (1) Set matrix D to be same as weighted adjacency matrix
- (2) Iterate from 1 to n to create D superscript matrices
- (3) and (4) Iterate over every single element in the matrix
- (5) For each element, check to see if current value for shortest path is the lowest, or if a new route that goes through k happens to be shorter
  - Choose the min of those two and update D[i, j] accordingly
- Complexity:
  - Time efficiency in $O(n^3)$
  - Space efficiency in $O(n^2)$
  - Same as Warshall

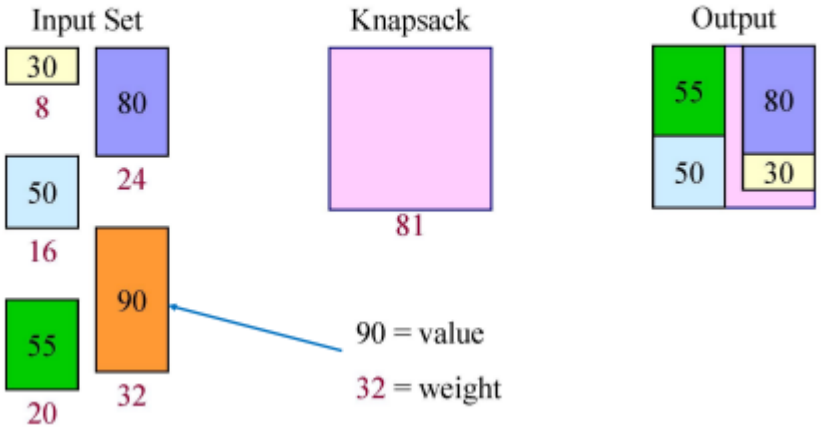
Extracting the Shortest Paths

- Dynamic Program often provides the value of the optimum but not other information
  - Needs to be extracted
- Predecessor pointers pred[i,j] can be used to find the optimal path
- As we build up the optimal path:
  - If shortest path from i to j passes through intermediate vertex k, we set pred[i,j] ← k
  - If it does not pass through any intermediate vertex, then we make pred[i,j] ← nil
- Having done that, we can find the path from i to j recursively:

- If pred[i,j] = nil the path is the edge (i,j)
  - This means there is directly an edge from one vertex to the other
- Otherwise find the path from i to pred[i,j] and from pred[i,j] to j

Knapsack Problem

- Fitting items with values and weights into a limited weight-capacity knapsack, such that value is maximized



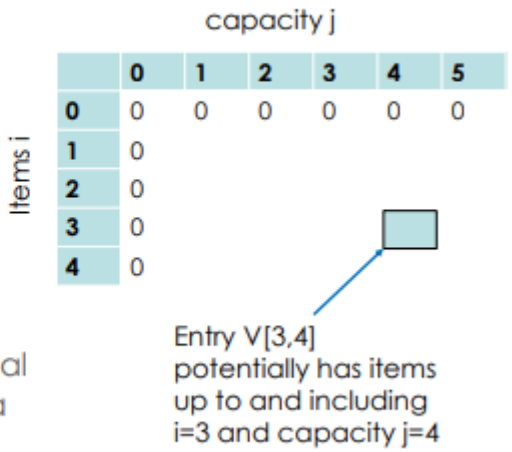
- Have a set of items which have two attributes (weight and value) and have container into which want to put them, such that containers carrying capacity isn't exceeded but want to pack in items for greatest value

Reminder: Principle

- Construct table of all subproblem results
  - Initialized entries of smallest subproblems
  - Remainder filled in following a precise order
    - Corresponding to increasing subproblems size
    - Use only entries that have already been computed
- Each entry is calculated exactly once
- Final value computed is solution to the initial problem
- Use iteration (and extra space for saved results)
  - Never recursion

Knapsack Solution Structure

- Sub-problem definition:
  - i items – $1 \leq i \leq n$
  - Weights – $w_1, \ldots, w_i$
  - Values – $v_1, \ldots, v_i$
  - Knapsack size j – $1 \leq j \leq W$
- Data Structure:
  - Create a table V of sub-problem solutions
  - Where V[i,j] is the value of the optimal subset of the first i items that fit into a knapsack of size j



Theory of Algorithms 30

- Have j – allowable weight of knapsack
- Will have n items, i = n, and j=w, at the max
  - Will create a table over i and j
- First row initialised to 0 – 0 items implies 0 weight
  - First column initialised to 0
- All possible items, up to and including item 3 and capacity of 4 for knapsack

Knapsack Recurrence Relation

- For entry V[i,j]
- Either don't include item i
  - (because it doesn't fit or isn't optimal)
  - Use previous optimal packing
  - i.e., V[i,j] = V[i-1,j]
- Or do include item i
  - Use optimal solution for knapsack that is smaller by the weight of item i
  - V[i,j] = V[i-1, j-w_i] + v_i

Initialisation:

$$V[0,j] = 0$$
$$V[i,0] = 0$$

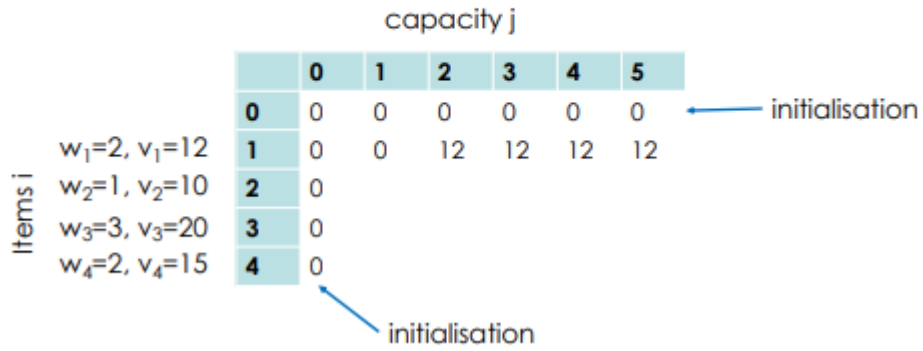
Recurrence:

$$V[i,j] = \begin{cases} V[i-1,j] & \text{if } j < w_i \\ \max(V[i-1,j], V[i-1,j-w_i] + v_i) & \text{else} \end{cases}$$

- Recurrence relation on each entry of the table – choice here is if want to look at previous solutions, can choose not to include the ith item (solution is optimal one to the same capacity for knapsack, with one item less) i.e. V[i, j] = V[i-1, j]
  - Or can include ith item – need to look at optimal solution with the weight of ith item subtracted, and add value of current item i.e. V[i, j] = V[i-1, j-w1] + vi
- Table stores the value of the current best solution for knapsack packing
- Recurrence relation:
  - Either provide previous solution if it so happens that the weight of the current item is larger than the current capacity (can't overflow the knapsack)
  - Otherwise look for max value by excluding item i, or including it

Knapsack Example: i=1

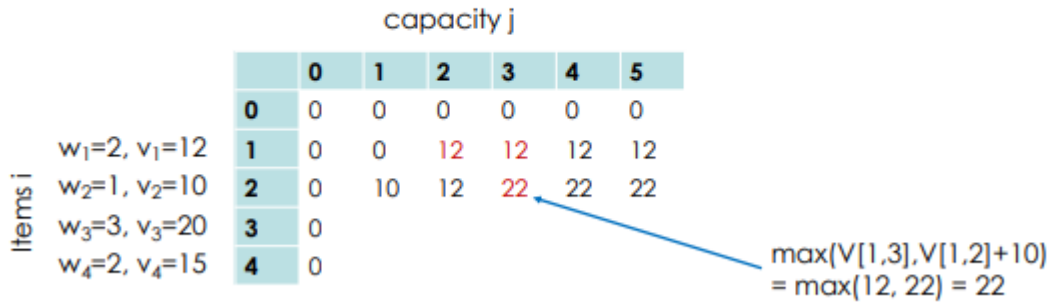
$$V[i,j] = \begin{cases} V[i-1,j] & if\ j < w_i \\ \max(V[i-1,j], V[i-1,j-w_i] + v_i) & else \end{cases}$$



- Initial values on the left – first item has weight = 2, value = 12
- What is the maximal value of the items up to and including current row with a weight capacity allowance in knapsack up to and including current column
    - E.g. row 1, column 1 says – take first item and put it in a knapsack with carrying capacity = 1, BUT since weight of first item =2, will use first option of recurrence relation i.e. use set that was derived without current item
    - E.g. row 1, column 2 – able to put an item in, so would be max(0 { from row 0, column 2}, previous best value which is [row 0, column 0] + current item which is 12)

Knapsack Example: i=2

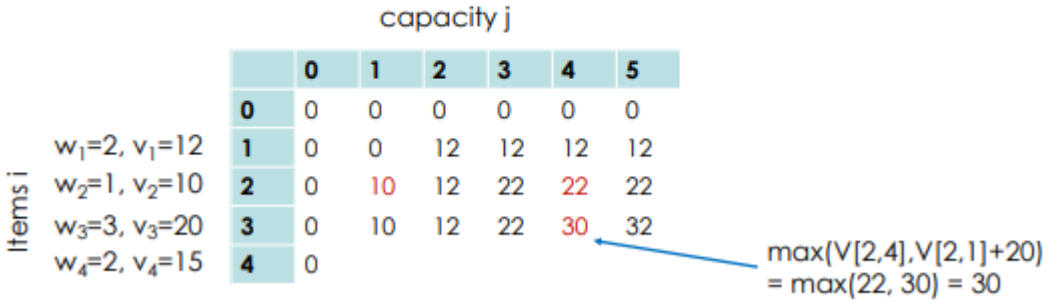
$$V[i,j] = \begin{cases} V[i-1,j] & if\ j < w_i \\ \max(V[i-1,j], V[i-1,j-w_i] + v_i) & else \end{cases}$$



- Now try to fill out row 2 – i.e. trying to put item 2 into the knapsack

Knapsack Example: i=3

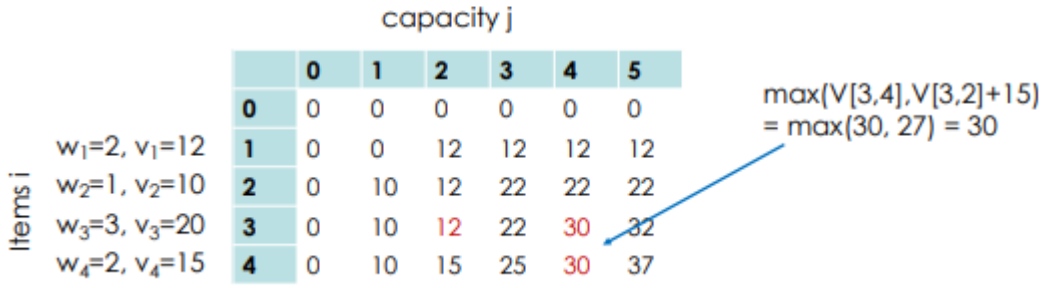
$$V[i,j] = \begin{cases} V[i-1,j] & if\ j < w_i \\ \max(V[i-1,j], V[i-1,j-w_i] + v_i) & else \end{cases}$$



	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0					

- Same procedure as previous – try to include item 3
- Will either check previous row, current column OR previous row, current column – current item weight (for the [3,4] example

$$V[i,j] = \begin{cases} V[i-1,j] & if\ j < w_i \\ \max(V[i-1,j], V[i-1,j-w_i] + v_i) & else \end{cases}$$

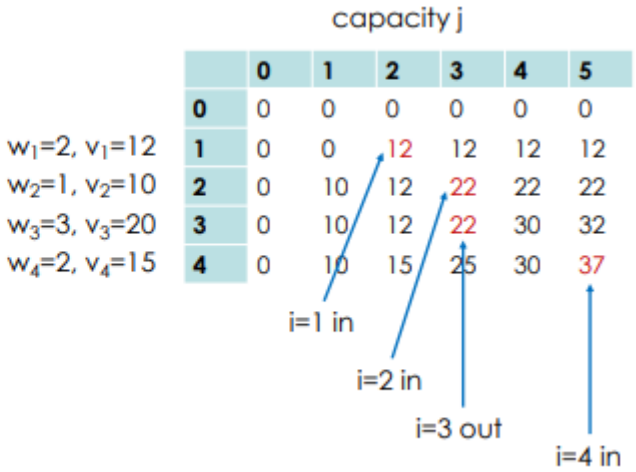


	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

- Same procedure as previous – find that not including item 4 is best

Knapsack Contents

- V[i,j] only provides knapsack values, not contents
- But we can work backwards
  - If $V[i-1,j] \neq V[i,j]$ then item i has been included, otherwise not
  - Then proceed accordingly to check $V[i-1, j-w_i]$ (*i* included), or $V[i-1,j]$ (*i* not included)



	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

- Need to find out what the items are – in order to do this, backtrack by starting in bottom right corner and comparing that value against the previous entry and also against the other it could have been selected from
- Example – start with 37, compare with 32 and 22
    - So, one row previous (same column) and then back by the amount of weight of an item
    - Since 37 != 32, have to include item 4
    - Then compare that to knapsack that was used to incorporate current item – which was 22
    - Compare 22 to previous row, which is also 22, thus, exclude item 3
    - And so forth

Memory Function

- Dynamic programming solves problems with a recurrence relation that provides overlapping subproblems
- Top down (brute force)
    - Solves common subproblem more than once, leading to exponential calculations – think of Fibonacci tree
    - Need a way to find the minimum number of table entries needed
- Bottom up (dynamic programming)
    - Fills table with all subproblems but calculates each one only once – but converging to final solution to overall problem
- Memory functions combine both
    - Exception to the rule of not using recursion in dynamic programming
    - Fill table with nulls, proceed top down and only calculate when needed
    - Equivalent to lazy evaluation

Knapsack Memory Function Solution

```
sack(i,j)
   if V[i,j] < 0 // -1 is null
      if j < w[i] then val ← sack(i-1,j)
      else val ← max( sack(i-1,j),
                      v[i]+sack(i-1,j-w[i]))
      V[i,j] ← val
   return V[i,j]
```

		0	1	2	3	4	5
	0	0	0	0	0	0	0
w_1=2, v_1=12	1	0	-1	-1	-1	-1	-1
w_2=1, v_2=10	2	0	-1	-1	-1	-1	-1
w_3=3, v_3=20	3	0	-1	-1	-1	-1	-1
w_4=2, v_4=15	4	0	-1	-1	-1	-1	-1

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0		12	22		22
3	0			22		32
4	0					37

- (1) Calculate knapsack value for element [i, j] in table
- (2) If entry is not initialised – used -1s to represent null, if -1, then calculate it
- (3) Ask if current weight of knapsack is capable of containing that particular item

    - If it turns out that knapsack is smaller than current item, use previous knapsack – so set the optimal value to the previous row of the same column
- (4) Otherwise, could potentially include item – them becomes maximal of (the previous knapsack or the knapsack that is smaller from previous row)
    - Here, doing a recursive step – calling the sack function at these points, with different, smaller values
    - Recurses backwards through the table
- (5) once have correct value from recursion, can then fill it in the current entry in table
- (6) At this point, will have correct value and will return that
    - Base case is when return the 0 value from the initialisation steps

Summary

- Dynamic Programming is typically applied to a recurrence relation
    - Break a problem down into smaller, more easily solved subproblems. To avoid solving these subproblems several times, their results are computed and recorded in a table.
    - Can only be applied when the principle of optimality holds
- The biggest limitation of Dynamic Programming is the number of partial solutions we must keep track of
    - If the objects are not ordered, we have an exponential number of possible partial solutions
- The memory function tries to combine the strengths of top-down and bottom-up by doing only those subproblems that are needed and only doing them once

Greedy Techniques

Introduction

- Greedy technique repeatedly tries to maximize the return based on examining local conditions, with the hope that the outcome will lead to a desired outcome for the global problem
    - i.e. Solve optimisation probs (have score that trying to maximise) – idea is that always grab best solution that you can at any given step, which should lead to overall best solution
- Examples:
    - Dijkstra's single-source shortest paths
    - Change making
    - Huffman coding

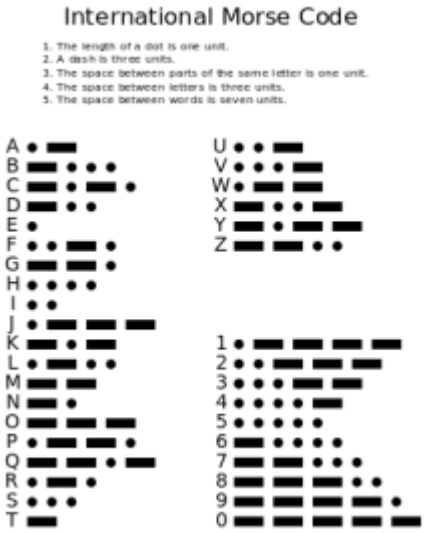
Principle

- Optimization problems solved through a sequence of choices that are:
    - Feasible – satisfy problem constraints
        - i.e. must be a "legal" and valid move in state space of problem
    - Locally optimal – best choice among all feasible options for that step
    - Irrevocable – no backing out
- A greedy grab of the best alternative, hoping that sequence of locally optimal steps will lead to a globally optimal solution
- Sometimes approximation is acceptable (local optimality waived)
    - i.e. even without fulfilling all three requirements stated above, can still apply greedy technique (if you're willing to accept an approximate optimisation)
- Not all optimizations can be solved this way

Change Making Revisited

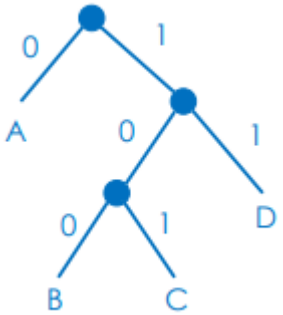
- Problem:
    - Give change for a specific amount n, with the least number of coins of the denominations d1 > d2 > … > dm
- Example:
    - Smallest change for R2.54 using R5, R2, R1, 50c, 20c, 10c, 5c, 2c, 1c
    - R2 + 50c + 2c + 2c = R2.54
- Algorithm:
    - At any step choose the coin of the largest denomination that doesn't exceed the remaining total;
        - Repeat
    - Optimal for reasonable sets of coins
    - Think of an example where it isn't optimal?

Text Encoding



- Has applications in text and also video encoding
- Assign bit patterns to an alphabet of characters so that in general texts fewer bits are used overall
    - Want to convert text into bit stream – but in such a way that the bit stream occupies the *least* amount of space possible
- Fixed length
    - Same number of bits for each character
    - E.g., ASCII
- Variable length
    - Number of bits for a character varies according to probability of occurrence
        - i.e. characters that occur more often in text stream should occupy less space (have fewer bits)
    - Leads to better compression
    - E.g., Morse's telegram codes

Huffman Trees



- Alternative to using a special character that demarcates between bit sequences
- How to determine which bits belong to which character in a variable length encoding?
- Solved by Prefix-free codes where not code is the prefix of another character
    - If have set of bits that belong to a character – then that set of bits doesn't occur in any of the other start sequences for characters
    - "Uniqueness" constrain

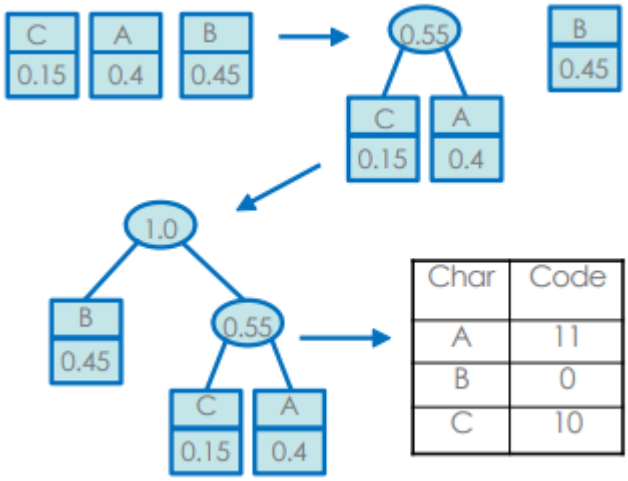
- Huffman Tree:
    - Implements prefix-free codes
    - Leaves are characters, left edges encode 0-bits, right edges 1-bits
    - Walk the tree to encode and decode – from root to leaf
        - Depth-first walk of Huffman tree to arrive at a particular code
    - Example encoding: ABADC → 0100011101
    - Example decoding: 11000101 → DAAAC

Huffman Coding Algorithm

```
1. Initialize n one-node trees labeled with the
   characters of the alphabet
2. Record the frequency/weight of the character in the
   root
3. REPEAT until a single tree is obtained:
   • Find two trees with the smallest weight
   • Make them the left and right sub-tree of a new
     tree and record the sum of their weights in the
     root
```

- Build a Huffman Tree given an alphabet and associated probabilities (for each particular character occurring)
    - Characters with higher probabilities have lower bit lengths
- Each node has symbol and probability attached to it
- Take best two probabilities and join them – repeat until everything is amalgamated into a single tree

Huffman Coding Example



Char	Code
A	11
B	0
C	10

- Look at two points with smallest overall weighting – so, C and A
    - These will appear further down into tree – doesn't matter which goes L or R, but be consistent
- Sum 0.4 and 0.15 and place in new root
    - Have new tree, with B available

- Now try to combine these sets of trees
    - Will combine B and 0.55 tree – smaller value goes left, large goes right (to be consistent)
    - If sum 0.55 and 0.45 = 1, implies that we've finished
- Going to the right = 1, left = 0

Notes on Huffman Coding

- Entire point is to have variable length scheme with text encoding that is more efficient i.t.o. space occupied – need to have some way of measuring this
    - Use compression ratio for this
- Compression Ratio:
    - Standard measure of compression
    - Denominator = previous representation
    - CR = 100 * (y - x) / y, where x is compressed, and y is uncompressed
    - Comparing fixed length vs variable length encoding
    - Typically, 20-80% in the case of Huffman Coding
- Yields optimal compression provided:
    - Probabilities of character occurrences are independent
    - And are known in advance

Summary of Greedy Techniques

- Strengths:
    - Intuitively simple and appealing
- Weaknesses:
    - Only applicable to optimization problems
    - Doesn't always produce an optimal solution

Course Conclusion

- Coverage:
    - Algorithmic strategies for solving computational problems
    - Including well-known solutions to common optimization, computational geometry and graph problems
- Key Skills:
    - Problem solving – Understand how to apply the different algorithmic strategies
    - Analysis – be able to determine the efficiency of a simple iterative or recursive algorithm
    - Knowledge and application – know the common algorithms under each strategy and be able to apply them to sample data