IFT 2125 Study Guide for Final!

Sean Grogan

# Asymptotic notation

Algorithms are classified according to their time complexity in the worst case, the average or best case

In the course, we became interested in complexity in the worst case

## Theoretical analysis

To calculate the time complexity of an algorithm, we count the number of elementary operations that must execute in the worst case

Asymptotic notation is used to express the complexity of an algorithm

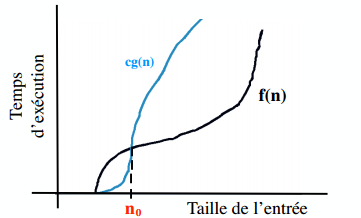
# Big

Let there be a function. We define the order of as as the set of bounded functions above by:

Threshold rule: Given two strictly positive functions then

The Big O relation is reflexive and transitive

If and then



The phrase signifies the rate of growth of is smaller or equal to the rate of growth of

We can use Big O notation to order functions by their rate.

## Properties of Big

Maximum rule:

Product rule

Transitive rule

# Big

Let there be a function . We define as the set of functions bounded below by :

Duality rule:

# Big

We say that is in the exact order of , denoted as if

This is equivalent to

# Limit Laws

If then or

If then and

If then and

# Conditional Asymptotic notation

There exists and

In a similar way, we can define

In a similar way we can define

# Smoothness Rules

A function is eventually non decreasing or END, if such that

A function is b-smooth for if

1. It is END

A function is smooth if it is b smooth for all

## Smoothness Rule:

Let be a smooth function, if it exists and if it exists is an END Function, then

# Solving Recurrences

Intuition or Conjecture:

Step 1: Calculate the first values of the recurrence

Step 2: see the patterns or regularities

Step 3: Find a general form

Step 4: Demonstrate the form is correct

# Solving homogeneous linear recurrences with constant coefficients

For the recurrence

Use these steps to solve:

1. Find the characteristic polynomial of the recurrence
2. Find the roots of

If the roots are distinct

1. The general solution is in the form
2. Solve the system of linear equations with the initial conditions to find the values of
3. Write the solution according to the constants

For the recurrence

Use these steps to solve:

1. Find the characteristic polynomial of the recurrence
2. Find the roots of

If the roots are NOT distinct

1. The general solution is in the form where we have roots of of multiplicity
2. Solve the system of linear equations with the initial conditions to find the values of
3. Write the solution according to the constants

# Solving of non-homogeneous linear recurrences with constant coefficients (special case):

For the recurrence

Where is a constant

Follow these steps to solve

1. We begin to transform the recurrence to a homogenous recurrence  
   for this particular case, we multiply the recurrence by and replace with in the obtained equation  
   Subtracting the initial recurrence, this new recurrence, we obtain a homogeneous recurrence
2. Solve (Homogenous case) with the as usual, but by ensuring that the initial conditions also satisfy the original equation:

# Changing of variables

Sometimes it is easier to start with a change of variables when we want to solve a recurrence

For example, if . By the change of variables we obtain the linear, non-homogenous recurrence

This recurrence is solved in the usual way and then the solution obtained is expressed in in the function of utilizing the substitution and therefore

# Greedy Algorithms

Easy to develop

A local optimum is selected regardless of the effects in the future (not in the reverse)

We would like this local strategy leads to a global optimum

We must prove the optimality to show that greedy algorithm finds the optimal solution

Examples in class:

* Money Change
* Minimal Tree Cut (Kruskal – Prim)
* Shortest Path (Dijkstra)
* Backpack
* Simple Waiting Queue

## Money Change

Problem:

It has an unlimited number of coins of different values. We want to make money in the amount so that returns the fewest pieces

### Greedy Solution:

We begin by giving the maximum coins of greatest value (local optimum) and then complete the amount n with parts smaller values

### Proof of Optimality:

The optimality here depends on the value of coins in our possession and the fact that we assume an unlimited number of each piece.

#### Python code

def makechange(n):

C = [100, 25, 10, 5, 1]

S = [] #solution Set

s = 0 #sum of the solution set

fin = False

while ((s != n)&(fin == False)):

x = max(C)

i = 0

while((s + x > n)&(i<len(C))):

x = C[i]

i += 1

if(i<len(C)):

fin = True

print('result not found')

S.append(x)#adds coin to the change

s = s + x

print(s)

print(S)

## Minimum Spanning Tree

### Problem:

We are given a connected, non-orientated graph a cost function, we want to find such that is a tree such that the sum

Is minimal

### Greedy solution:

1. Start with an empty set of edges at each step and select the smallest edge cost that has not yet been chosen or rejected (regardless of where it is in the graph) (Kruskal)
2. Start vertex in a graph and construct a tree from the top of each step by selecting the edge which adds minimal cost to the existing new tree node (Prim)

### Kruskal

The algorithm maintains a forest of trees

At each iteration, we choose the edge with the minimal cost

This edge is accepted, if it reads two different sub-trees, otherwise it is rejected (could form a cycle)

The algorithm is completed when a single tree is left

#### Pseudo Code

Function Kruskal(G = {N, A}: graph; length: A->R+): set of edges

{initialization}

Sort A by increasing length

n <- the number of nodes in N

T <- [] #empty set

Initialize n sets, each containing a different element of N

{Greedy Loop}

Repeat

e <- {u,v} <- shortest edge not considered

ucomp<-find u

vcomp<-find v

if ucomp != vcomp

merge(ucomp,vcomp)

T<-T Union {e}

Until T contains n-1 edges

Return T

### Prim

We choose a random vertex in a "cloud" and the minimum spanning tree constructed by enlarging the "cloud" one vertex at a time.

We keeps track each vertex , a label , which here is equal to the minimum weight among the weights of the edges connecting v to a vertex inside the cloud.

At each step

* We add to the cloud the outer vertex with the smallest weight
* We update the labels of vertices adjacent to

#### Pseudo code

Function Prim(L[1..n, 1..n]): set of edges

{initialization: only 1 node is in B}

T = [] {will contain the edges of the minimum spanning tree}

For i = 2 to n:

Nearest[i] <- 1

Mindist[i]<-L[i,1]

{greedy loop}

Repeat n-1 times

Min<-ininity

For j<-2 to n do

If 0<=mindist[j]min:

min<-mindist[j]

k<-j

T<-T Union {{nearest[k],k}}

Mindist[k]<- -1 {add k to B}

For j<- 2 to n:

If L[j,k]<mindist[j]:

Mindist[j]<- L[j,k]

Nearest[j]<-k

Return T

### Optimality

#### Property partition

* Consider a partition of the vertices of G into two sets U and V
* Let e ​​be an edge of minimum weight between U and V
* So, there exists a minimum spanning tree of G containing e

#### Proof:

* Let T be a minimum spanning tree of G
* If T does not contain e, C is the cycle formed by adding e to the tree T and let f be an edge between U and V
* Through the domain of cycles, we have that
* As we took e minimum weight, if there is where and then we obtain another MST replacing f by e

The optimality of both algorithms follows from the property of partition of MSTs

#### Kruskal

The graph of the partition is considered here, since an edge (u, v) of minimum cost, on the one hand all the vertices belonging to the connected component of the other of u and all other vertices.

If u and v are not part of the same connected component, the property partition guaranteed that the edge (u, v) is part of a MST

#### Prim

Here, the partition is considered cloud / non-cloud

## Shortest Path

Problem: Let be a connected, non-orientated (or orientated) graph and a cost function. Given a source vertex, we want to find the shortest paths from the source to all other vertices.

Greedy Solution: Dijkstra algorithm

### Dijkstra Algorithms

* The distance between a vertex v and a vertex s is the minimum total weight of a path between 'v' and 's'
* Dijkstra's algorithm is a greedy algorithm that calculated the distance between a vertex 'v' and all the other vertices of a graph
* Here we assume
  + The graph is connected
  + The weight of the edges are non-negatives
* We're going to grow a "cloud" of vertices, initially containing 'v' and eventually covering all vertices
* We will give a label and 'd ​​(u)' at each vertex, representing the distance between 'v' and 'u' in the subgraph is vertices in the cloud and adjacent edges
* At each step:
  + We add a vertex u to the cloud that is outside the could that has the smallest label ‘d(u)’
  + We update the labels of vertices adjacent to u

Problem: Let be a connected, non-orientated (or orientated) graph and a cost function. Given a source vertex, we want to find the shortest paths from the source to all other vertices.

Greedy Solution: Dijkstra algorithm

Proof of Optimality: comes from the following proposition

Proposal: With Dijkstra, whenever a vertex u is included in the cloud, D (u) is the minimum cost path between u and the source and vertex path is entirely within the cloud

#### Pseudo code

Function Dijkstra(L[1..n,1..n]):

Array[2..n]

Array D[2..n]

{initialization}

C<- [2, 3,...,n] {S=N\C exists only implicitly}

For i<- 2 to n do D[i]<-L[1,i]

{greedy loop}

Repeat n-2 times

v<-some element of C minimizing D[v]

C<-C\{v} {and implicitly S<-S Union {v}}

For each w in C do:

D[w]<-min(D[w],D[v]+L[v,w])

Return D

## Backpack Problem

Problem: we have available objects of positive weights and have positive values . Our backpack has a capacity (maximum weight) of .

Our goal is to fill the bag so to maximize the value of the objects in the bag while respecting the constraint weight.

To be able to solve problems with a greedy algorithm, we suppose we can bring a fraction of each object for each object.

Goal:

Greedy solution:

Select each object in turn in a certain order, to the largest possible fraction of this object in the bag (without exceeding the weight capacity of the bag) and stop when the bag is full.

We select the object in the value per unit weight is maximum

Proof of optimality: comes from the following theorem

Theory: If objects are selected in descending order of then this strategy returns an optimal solution

### Proof of optimality

If the objects are chosen in descending order , therefore the greedy algorithm will produce an optimal solution. Without loss of generality, assume that the objects are numbered in descending order of their value per unit of weight i.e. that

Let be a solution found by the greedy algorithm. If all the (all the object are present in the backpack) the solution is clearly optimal. Otherwise, if is the smallest index such that (or and ). We have while and and . That is to say , the value of the solution X. We show that this value is V (X) is optimal.

Such that or any other solution is , the value of the solution. Since Y is the solution, which was

Now we have

We must show that is always to ensure optimality of and therefore the solution .

Therefore is smallest where (or ):

* If , therefore and . Therefore, for the case, we have
* If , therefore and . Therefore, for the case, we have
* If , therefore and . Therefore, for the case, we have
* , therefore and

So we show that , where is the smallest inex where (or ). We therefore write

Therefore,

And is optimal

## Simple waiting queue – Not in slides

We have a server with clients

Each client needs a service time

If the client must wait, its waiting time is denoted

Problem:

Theorem: If customers are served in ascending order of time of service requested, the greedy algorithm to find an optimal solution.

Proof of Optimality: Assume that an optimal solution exists where the order of customer service is different from the ascending time of service requested and derives a contradiction

### Proof of optimality of the greedy algorithm for the queues

Let be arbitrarily ordered client and services in the order . The total time took to serve customers in this order is

We will prove by contradiction. Suppose that is optimal and suppose that the order is not an order in which customers are served in ascending order of service time while in the order , such that and . If we change the positions of clients a and b to obtain a new order . To this order, we have

But then:

And therefore T(C) is not optimal. We have a contradiction.

# Dynamic Programming Algorithms

This approach is a bottom upwards approach

The idea is to solve a problem, we begin by solving smaller sub-problems and retains the values ​​of the sub problem in a dynamic programming table. These values ​​are then used to calculate the value of sub-problems of ever larger, until the solution to our global problem.

Principle of optimality: The optimal solution to a problem consists of optimal solutions to subproblems

Examples in class:

* Money return
* Backpack problem
* Shortest paths
* Sequence alignments

## Return Money

Problem was the amount and the pieces of value . We want to find integers such that

Solution: We will build a dynamic programming table where is the number of minimum number of pieces to produce the exact amount using only parts values

The optimal solution will be .

Table initialization we return 0 pieces for the money 0

To calculate , we have 2 choices,

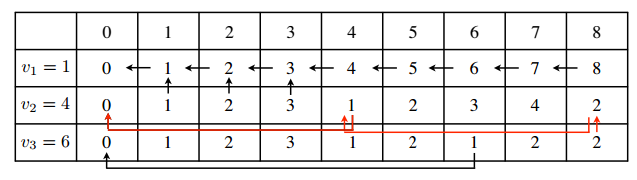
1. We do not take parts value V even though we now have the right. If you make this choice:
2. It takes at least a piece of value . In this case, after giving this piece to the client, the total remaining to be paid is . So if we made ​​this choice:

We want to minimize and therefore we pose:

It therefore has the following equations:

As the problem is not defined for and when , if we put these equal to infinity not consider the minimum. Also, if and , we set to mean that it is impossible to pay the amount with parts value.

Suppose we have a total of 8 pieces and values



## Backpack Problem

Suppose we have n objects of positive weights and positive values. Our bag as a capacity of maximum weight W

Goal:

We build a dynamic programming table where is maximum value of objects that can be transported if the maximum weight allowed is and objects that can be included are those numbered

The optimal solution is

Table initialization

To calculate we have 2 choices:

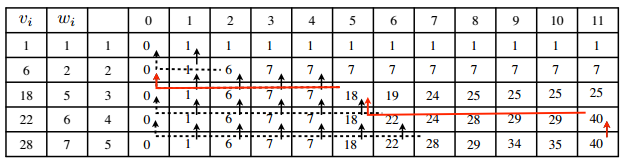
1. It does not add the item i in the bag. If you make this choice:
2. It does add the item i in the bag. If you make this choice:

We maximize

We therefore use the following equations

The problem is not defined for and if . We set to mean that if no object then the value is 0 regardless of the capacity and if because we do not want to consider negative weights.

Suppose we have 5 items 1,2,5,6 and 7 weight and values ​​1, 6, 18, 22, 28 and the capacity of our bag is 11.



Now, if we have the same problem but instead of objects, each one time, we have types of objects, each object in unlimited:

In this case,

1. is it does not take an object of type
2. is added to a bag type object for the first time
3. is added to the bag an object of type and there is already at least one object of this type in the bag

The dynamic programming becomes:

## Shortest Path:

Problem: Let G be a directed graph, N all these vertices and A the set of these edges. Each positive edge to a weight representing the distance between the two vertices. We want to calculate the length of the shortest paths between all pairs of vertices of G.

Solution: we build matricies for

Where is the length of the shortest path between and and for which the intermediate peaks are generally

The length of the shortest path between each pair of the vertices is

Initialization: we have (because the distance between a vertex and itself is silly…). For an edge between I and j, is the weight of the edge. If an edge does not exist, the weight is infinity.

To calculate , we have 2 choices:

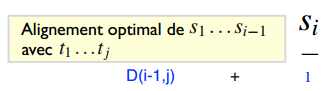
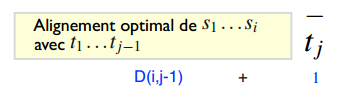
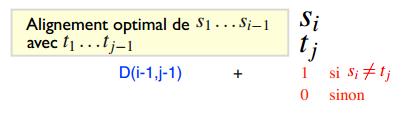
1. It does not pass through :
2. It passes through :

We minimize and therefore we show:

## Alignments

Given two sequences and , we define as the edit distance between the prefix of size of and the prefix of size of . We define a matrix of size called dynamic programming matrix. The idea is then to express D (i, j) based on the values ​​of D for smaller that pairs of indices (i, j).

Calculate from the three cases: , and :

1. The alignment is ended with the removal of   
   
2. The alignment ends with the insertion of   
   
3. The alignment ends with the alignment of with   
   

Building the table:

Initial conditions:

The recurrance relation for :

Complexity: To fill each cell of the table, we examine three cases. There are O (nm) cells and therefore time complexity of O (nm)

During filling of the table pointers keep:

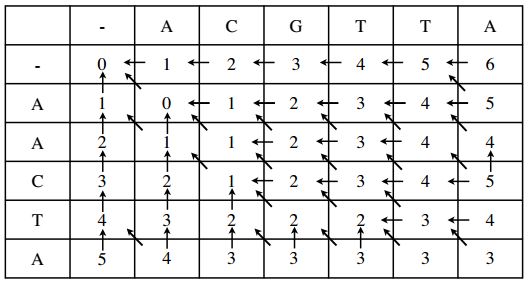
* of to if
* of to if
* of to if

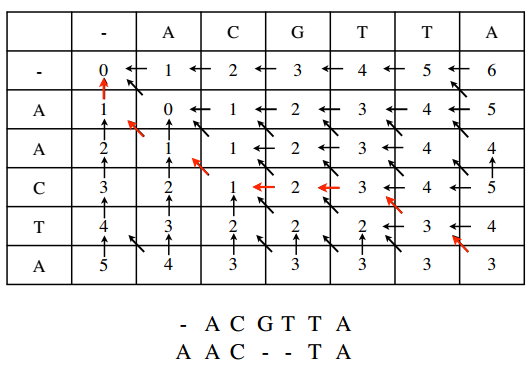
Optimal alignment: Start box and follow pointers to the case . A box can contain several pointers: several possible optimal alignments

Example:

Claculate the global alignment for the words ACGTTA and AACTA:

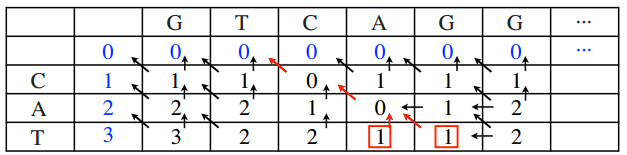
The recurrance relation for :





## Search approached a pattern:

Problem: We have a "small" pattern P of size m and a "long" sequence T of size n and we want to find all approximate occurrences of P in T (less than k errors).



* Initialize the first row to 0, and the first column as usual
* Same recurrence relations for the global sequence alignment
* Look at the line m all boxes containing values ​​smaller or equal to k
* To find an alignment, follow the pointers to the first line

### Local alignment - Smith-Waterman algorithm

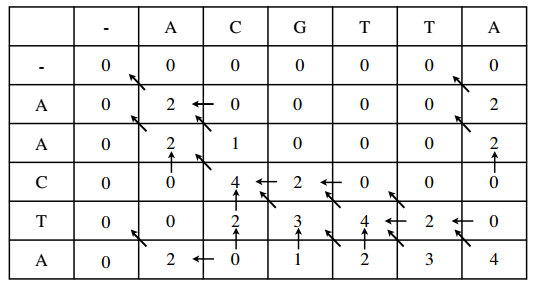
Recurrence relations:

0 in the recurrence will ignore any number of characters at the beginning of sequence

To find a local maximum alignment score:

* We filled the table
* We are looking for a box containing the maximum value of the table
* In this case c, we follow the pointers to a box containing the value 0

Example: calculate the local alignment for the words ACGTTA and AACTA:



# Divide and Conquer

This approach is a top down

The idea is that we take a problem and break it into sub-problems. Sub-problems are solved (possibly by breaking again) and the results are combined to obtain the solution to the original problem.

General Construction: If the instance is small enough to solve a conventional algorithm is used to solve, otherwise it is broken into pieces (if possible of the same size) and recalls the algorithm on these pieces and then recombine.

Examples from class:

* Binary Research
* Merge sort
* Rapid search
* Search
* Median

3 requirements for an algorithm to divide-and-conquer effective:

1. While deciding when to use the simple algorithm on small bodies rather than recursive calls
2. The decomposition of an instance in sub-instances and recombination of sub-solutions must be effective
3. The sub-bodies should, as far as possible, about the same size

Often the time complexity of an algorithm divide-and-conquer on an instance of size can be written as:

Where is the time took to break and rebuild.

If it can be shown that for a certain then the following theorem automatically gives us the complexity of the algorithm:

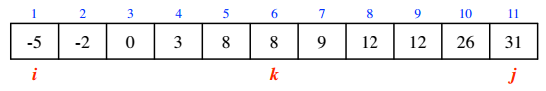
Let is a function eventually non decreasing such that

Where is a power of . So,

## Binary research

Problem: Let a list of integers sorted in ascending order i.e. we have . The problem is to find an integer in the list, or if the whole is not in the list, find the position where it should be inserted.

Example: Find 7 in the following sorted list:



The pointer i and j to 1 in size (L) is initialized and if we look at 7> L [j]. If so, the insertion position 7 is n +1, we do binary search.

Binary search: if i = j, i return, otherwise seek to compare the element floor ((i + j) / 2) and continue the search in the right subarray

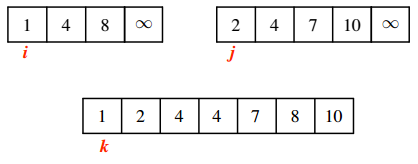
## Merge Sort

Problem: Sort a list of integers L [1 .. n]

Tri-merge Solution: Divide the list into two. Sort parts by recursive calls and merge solutions of each party being careful to maintain order.

Complexity: where is the pairs, where is the time to merge in . therefore here, we have

Merge:



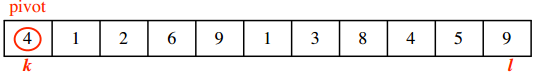
## Rapid sort

Problem: Sort a list of integers L [1 .. n]

Solution Quick Sort: Take the first element of the list as pivot, pivot the L around this pivot so that the smallest elements are in a list L1 to the left of the pivot and the other elements in a list L2 right the pivot. Sort recursively L1 and L2 with Quick Sort

Complexity: on average . In the worst case,

Pivoting:



* Two pointers k initialized to 1, initialized to the size of .
* Move to the right until element pivot
* Move the left until a pivotal element
* Exchanging and and repeat until
* Exchange pivot and

Complexity: on average . In the worst case,

Can we avoid the worst case?

yes

* The median of the list in linear time is calculated using the algorithm implemented with the selection pseudo-median procedure
* A pivoting between the different list L was used in three parts: the smaller the pivot elements, the elements equal to the pivot elements and larger than the pivot
* Quicksort is recalled only on lists of smaller and larger elements than the pivot

But no, The modified algorithm in this way still works in O (N log N) but the constant hidden due to this change is so great when the new practice quicksort is becoming slower than merge sort.

## Selection Problem

Let L [1 .. n] a list of integers, and let s be an integer between 1 and n. The s-th smallest integer L is defined as the element that would position s if the list L was ordered but not decreasing.

Given L and s, the problem of finding the s-th smallest element of L is called the selection problem.

* An algorithm, median (L), which returns the median of the list L
* The algorithm called L will be partitioned into 3 sections:
  + L[1..k] that contains the elements < p
  + L[k+1..l-1] that contains the elements = p
  + L[l..n] that contains the elements > p

Can we modify the algorithm selection for not using the median?

Yes - a pseudo-median is calculated

Algo PseudoMed(L[1..n])  
If n<= 5:  
 Return trueMedian (L)  
else:  
 z <- floor(n/5)  
 M <- [1..z]  
 for i in 1 to z:  
 M[i] <- trueMedian(L[5i-4:5i]

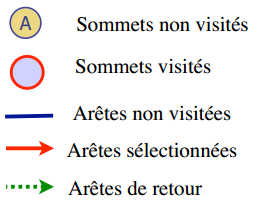
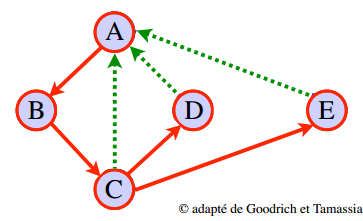
Fin for  
Fin if  
return (M, ceiling(z/2))

It can be shown that the selection algorithm (L [1 .. n], s) using the pseudo median is

So to calculate the median this selection algorithm is used with selection (L[1..n], ceiling(n/2))

# Graphs

## Depth-first traversal

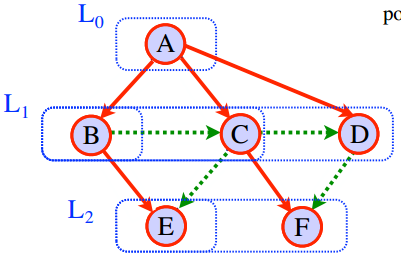
 

Algo DFSearch(G)  
 for all v in N:  
 mark[v] <- not visited  
 for all v in N:

If mark[v] is not visited:  
 dfs(v)

Algo dfs(v)  
 mark[v] <- visited  
 for each node w adjacent to v:  
 if mark[w] = not visited  
 dfs(w)

## breadth-first traversal



Algo BFSearch(G)  
 for all v in N:  
 mark[v] <- not visited  
 for all v in N:  
 if mark[v] = not visited  
 bfs(v)  
Algo bfs(v)  
 Q <- Empty Queue  
 mark[v] <- visited  
 Q.enqueue(v)  
 if len(Q) is not 0:  
 u = Q.dequeue()  
 for each node w adjacent to u:  
 if mark[w] = not visited  
 mark[w] <- visited  
 Q.enqueue(w)

|  |  |  |
| --- | --- | --- |
| Applications | DFS | BFS |
| covering forest, connected components | OK | OK |
| paths between two nodes, cycles | OK | OK |
| Shortest Path |  | OK |

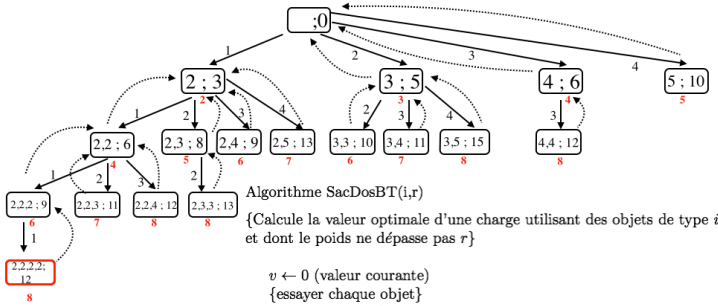
# Backtracking algorithms

The idea is to browse an implicit tree all sub-solutions to a problem to find the optimal solution

A backtracking algorithm begins at the root of this tree implicit (null solution) and explores in depth the tree nodes and constructing partial solutions progressively.

Classes Examples:

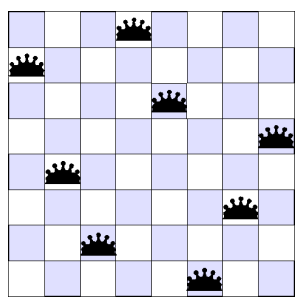
* Backpack
* 8 queens



Algo BACKPACK(I,r):  
 {Calculated the optimal value of a load using objects of type I to N and whose weight does not exceed r}  
 v<-0   
 for k in I to n:  
 if w[k] <= r:  
 v<-max(v, v[k] + BACKPACK(k, r-w[k])  
 return v

## 8 Queens

Problem: Place 8 queens on a chessboard so that no queen threatens another. (A queen threatens all parts positioned in the same row, the same column or the same diagonal).



Our 8 queens problem can be represented by a tree A = (N, E) where

N is the set of vectors k-promising

A vector V is k-promising, if for all pairs of integers i, j between 1 and k were

E is the set of edges such that if and only if such that

* u is k-promising
* v is k+1-promising

Algo QueenBT(sol, k, col, diag45, diag135)  
if k=8  
 retuen sol  
else  
 for j in 1 to 8  
 if j is not in col and   
 j-k is not in diag45 and   
 j+k is not in diag135  
 sol[k+1]<- j  
 QueenBT(sol, k+1, colU{j}, diag45U{j-k}, diag135U{j+k}

We call the algorithm with QueenBT([], 0, {},{},{})

# Branch and Bound

The idea is to browse an implicit tree all sub-solutions to a problem to find the optimal solution

At each node, a terminal is calculated for values ​​of the solutions resulting from this node

If this marker shows that these solutions will necessarily be worse than the best solution found so far, we do not need to explore this part of the graph

Class:

* Summons
* Backpack

# Probabilistic Algos

The main characteristic of a probabilistic algorithm is that the same algorithm can behave differently when it rolled several times on the same data. The execution time and the same result can vary considerably at each execution

Two types of probabilistic algorithms do not guarantee the accuracy of their results:

* numerical algorithms
* Monte Carlo algorithms

A probabilistic algorithm that always returns the correct result is called Las Vegas

1. Numeric: Returns an approximate solution to a numerical problem  
   More time = More Precision
2. Monte Carlo: Always returns an answer but may be wrong.  
   More time = greater probability that the response is good
3. Las Vegas: Never returns an incorrect answer but sometimes finds no response at all  
   More time = greater probability of success in each instance of departure

Expected time: Defined on each instance

Expected time(w) = This is the average time it takes a probabilistic algorithm to solve a large number of times

## Numeric Algos

These are the first algorithms to use random

Returns an approximate solution to a numerical problem

The returned solution is always rough, but its accuracy increases with the time available to find a solution.

Usually, the error is inversely proportional to the square root of the amount of work done.

One of the first use of probabilistic algorithms: Estimate

## Las Vegas

Las Vegas type 1: Use the chance to guide his choices and always manages to solve the problem  
bad choice => slower; example, sort

Las Vegas type 2: Use the chance to try to solve a problem exits to failure  
bad choice => admits failure; example, 8 queens

Particularly useful when a deterministic algorithm exists for the problem is:

* Good average
* Bad to worst case

A Las Vegas algorithm for this problem can then:

* Eliminate the worst case instances
* Standardizing bodies
* hopefully maintain a good time

## Monte Carlo

Can be wrong (no warning if this is the case)

Find a correct solution with a high probability and regardless of the instance of departure

We say that a Monte Carlo algorithm is correct if it returns a p-correct solution with probability p or more,

An interesting property of the Monte Carlo algorithms is that it is often possible to reduce the probability of error by increasing the computation time (ie run the algorithm many times on each instance of departure)

Amplify the stochastic advantage