JICSCI803
Algorithms and Data Structures
September to December 2020

Highlights of Lecture 11

Advices on Some Basic Knowledge

Dynamic Programing

Change base 10 into base 2 – Examples

How to convert 49 in base 10 into a number in base 2?

```
49/2=24   remains 1
24/2=12   remains 0
12/2=6    remains 0
6/2=3     remains 0
3/2=1     remains 1
1/2 gets 0 remains 1
```

Change base 10 into base 2 – Examples

How to convert 0.625 in base 10 into a number in base 2?

0.625	
0.625*2=1.25	get 1
0.25*2=0.5	get 0
0.5*2=1	get 1

So 0.625=(0.101)B

Overflow errors

– The finite size of the **exponent** part of a floating point number means that we can only represent numbers with a **maximum size** related to this.

– Using 16 bit floating point numbers as an example:
  - Internal representation 0111101111111111
  - Sign bit is 0 so the number is positive
  - Exponent is 11110 = 30 –(bias of 15) = 15
  - Significand is 1.1111111111 (leading bit is implied)
  - So the number is $(1+1023/1024)^{15} \times 2 = 65504.0_{10}$
  - If we multiply this number by 2 the exponent is too big to store – we have an overflow.

Underflow errors

– Similarly, we can only represent numbers with a ***minimum size*** related to the size of the **exponent** field.

– Using 16 bit floating point numbers as an example:

- Internal representation 0000010000000000
  - Sign bit is 0 so the number is positive
  - Exponent is 00001 = 1 –(bias of 15) = –14
  - Significand is 1.0000000000 (leading bit is implied)
  - So the number is (1) $\times 2^{-14}$= $0.00003051757812 5_{10}$
  - If we divide this number by 2 the exponent is too small to store –we have an underflow.

Rounding errors

– These arise because we have ***a finite number of bits*** in which to store the significand.

• Consider (16 bit)

0011111000000000001 × 0100001000000000

= 1025/1024 × 3

= 3075/1024

– If we convert this to 16 bit floating point we would need 11 bits to store the significand.

– Because we only have 10 bits the final result is stored as 3076/1024 – we have a rounding error.

Root Finding

- We often need to find a value of *x* for which a function takes the value 0.
- Such *x* values are called the roots of the equation.
- For example the roots of the order 2 equation

  $f(x) = x2–5x+ 6$

  are

  $x= 2$ and $x= 3$.
- (Note: a0= 6, a1= 5, a2= 1)

Root Finding

– For order 2 equations we can find the root directly using the well-known quadratic formula

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

– Where a is a_2, b is a_1 and c is a_0.

– If the order is greater than 2 or the function is not a polynomial, finding a root may be much harder.

Numerical Root Finding

- Assume we have some function *f*(*x*) and we wish to find a value of x for which f(x) = 0.
- We can approximate such a root by using an iterative process.
  - This can be done in a number of ways.
  - We will consider 2 such ways
    - The interval bisection method
    - The ***regula falsi*** method
  - Both techniques have a common starting point.

Numerical Root Finding

– If we have 2 values of x, x_1 and x_2 such that $f(x_1) < 0$ and $f(x_2) > 0$ then it should be obvious that some value of x, x_r between x_1 and x_2 must be a root of the function.

– (Note: x_1 does not have to be less than x_2).

– We can use this as the basis of our root finding algorithm.

– Let us assume that we have a function f_eval(x) already defined which returns the value of f(x).

The interval bisection method

– Given x_1 and x_2 as already defined we calculate the value of *x* half way between them x_{mid}.

$$x_{mid} = 0.5(x_1 + x_2)$$

– If we evaluate $f(x_{mid})$, three possibilities exist:

i. $f(x_{mid}) = 0$ and we have found our root

ii. $f(x_{mid}) < 0$ and a root must lie between x_{mid} and x_2

iii. $f(x_{mid})$, > 0 and a root must lie between x_{mid} and x_1

– In cases *ii* and *iii* we can replace one of starting values with the midpoint value and try again.

– Each iteration will bring us closer to the root.

The interval bisection method in code:

```
function b_root($x_1$, $x_2$)
   $f_1$ = f_eval($x_1$)
   $f_2$ = f_eval(x2)
   repeat
      $x_{mid}$ = $(x_1 + x_2) / 2$
      $f_{mid}$ = f_eval($x_{mid}$)
      if ($f_1 * f_{mid} > 0$) then
        $x_1 = x_{mid}$
        $f_1 = f_{mid}$
      else
        $x_2 = x_{mid}$
        $f_2 = f_{mid}$
      endif
   until $f_{mid}$ is close to 0
   return $x_{mid}$
end
```

Stopping the process

– In practice, we almost never get a value of x_{mid} for which $f(x_{mid})$ is exactly 0.

– This is why the code on the previous slide used the test

```
until f_mid is close to 0
```

to terminate.

– This is usually a test based on some pre-set tolerance which will depend on how close to the correct answer we need to get.

– The actual code is usually something like

```
until abs(f_mid) < tolerance
```

Dynamic Programing

Dynamic Programming

Dynamic Programming (DP) is a problem solving strategy that is:

- General;
- Efficient;
- Easy to understand.

It is applicable to a wide range of different problems.

It usually finds a solution in polynomial time…

- … this is a GOOD THING™.

It is often the only efficient strategy (technique) we know for a problem.

Dynamic Programming Strategy

The simplest way to think about dynamic programming is to look at it as "clever brute force".

That seems to be a contradiction:

- Brute force is just looking at every possible solution;
  - Traversing the entire problem graph/tree;
- There is nothing clever about that!

Another way to look at it is that we:

- Break the problem into sub-problems;
- Re-use the solutions to the sub-problems.

We can best see how DP works by looking at some examples.

Dynamic Programming Example I: Fibonacci Numbers

We are all familiar with the Fibonnaci numbers:

1, 1, 2, 3, 5, 8, 13…

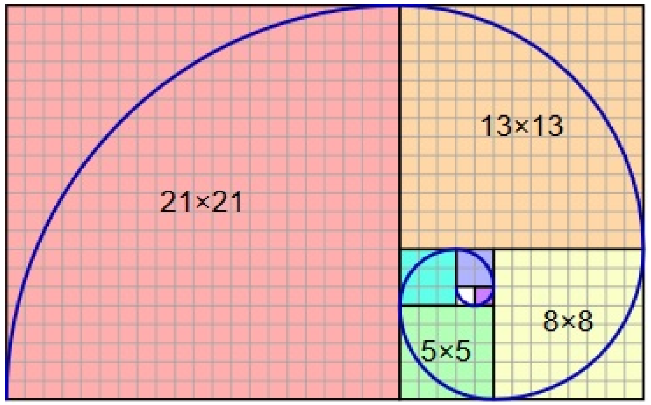
Each number is defined as the sum of its two immediate predecessors:

$\text{Fib}_1 = \text{Fib}_2 = 1$;

$\text{Fib}_n = \text{Fib}_{n-1} + \text{Fib}_{n-2}$, otherwise.

Dynamic Programming Example I: Fibonacci Numbers

The Fibonacci spiral: an approximation of the golden spiral created by drawing circular arcs connecting the opposite corners of squares in the Fibonacci tiling; this one uses squares of sizes 1, 1, 2, 3, 5, 8, 13 and 21.



We can compute Fibonacci numbers directly from this definition:

Recursive Fibonacci

```
Procedure fib(n: integer): integer
    f: integer
    if (n≤2) then
        f = 1
    else
        f = fib(n-1) + fib(n-2)
    fi
    return f
End procedure fib
```

This procedure is correct but it is not efficient.
It is, in fact, an exponential time algorithm.

Recursive Fibonacci a BAD THING™

From the code we can see that the time required to compute the nth Fibonacci number:

$T(n) = T(n-1) + T(n-2) + O(1)$

$T(n) > T(n-2) + T(n-2)$

$T(n) \in \Theta(2^{n/2})$

Interestingly, the time taken to compute the *n*th Fibonacci number is proportional to the *n*th Fibonacci number.

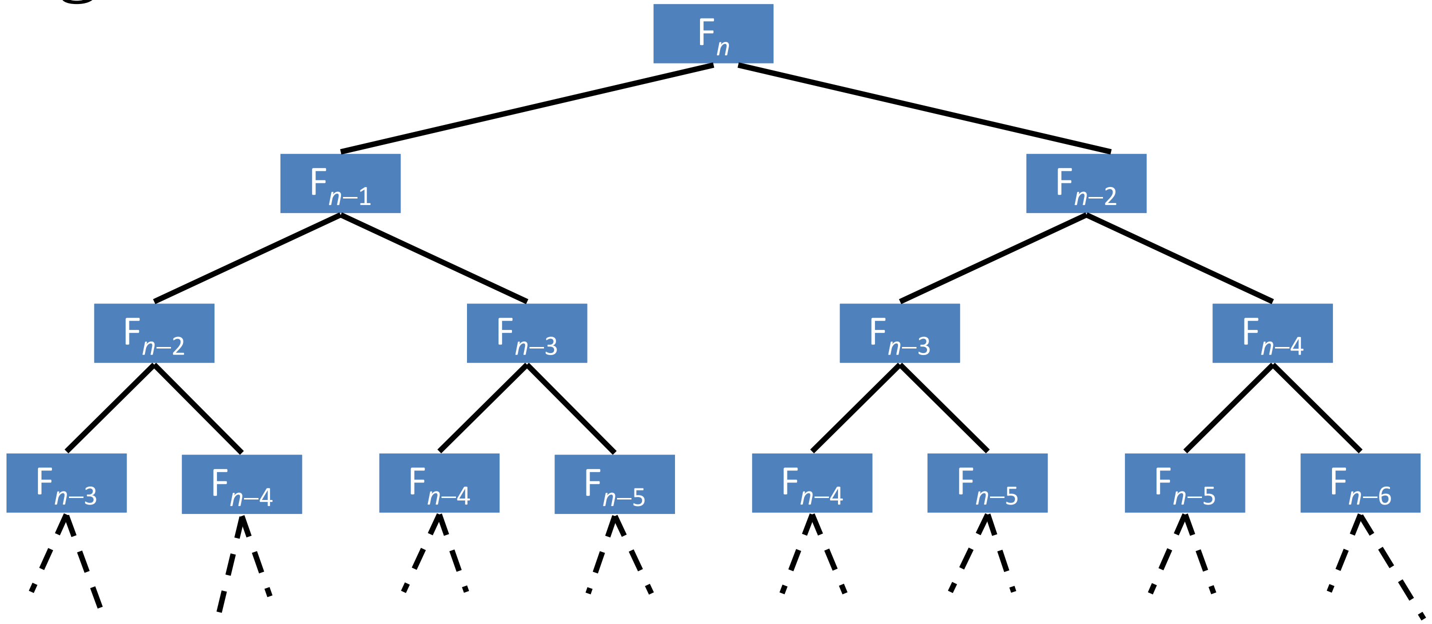
This is a bit like having a 1:1 scale map:

Accurate but hard to fold up.

Further Analysis

Let us look at this another way

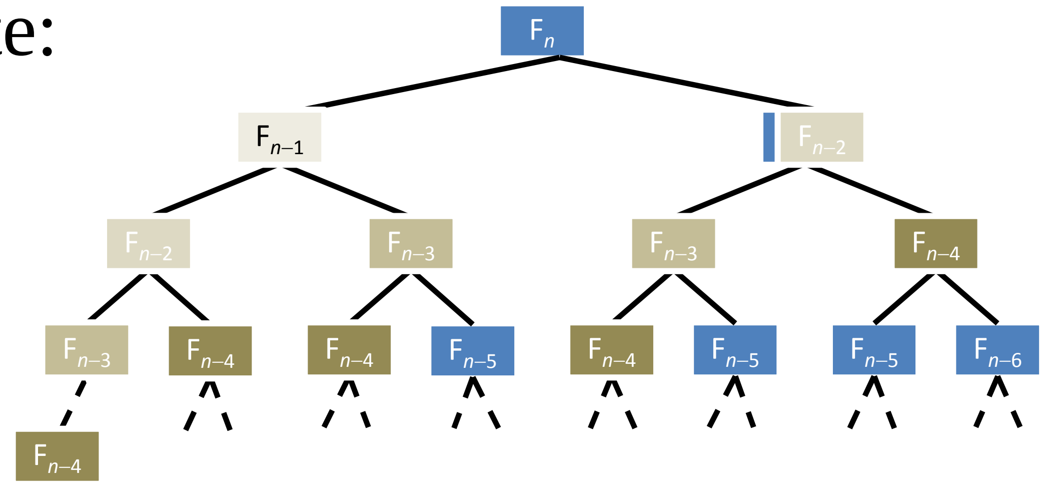
Evaluating F_n requires that we evaluate the following tree:



Further Analysis

So, to get F_n we evaluate:

- F_n once;
- F_{n-1} once;
- F_{n-2} twice;
- F_{n-3} three times;
- F_{n-4} five times;
- Etc.



The cost is in the repeated evaluations of the same thing.

What if we only evaluated each of them once?

This is the key insight in Dynamic Programming!

Memoization: the Heart of DP

The recognition that we only need to perform a given calculation once is central to Dynamic Programming.

How do we remember the previous evaluations?

We use a dictionary;

A hash table.

Let us look at the DP version of our fib procedure…

Recursive Fibonacci with Memoization

```
memo: dictionary = {}
Procedure fibDP(n: integer): integer
        f: integer

    if (n in memo) return memo[n]
    if (n≤2) then f = 1
    else
        f = fibDP(n-1) + fibDP(n-2)
    fi
    memo[n]=f
    return f
End procedure fibDP
```

Analysis

Now:

We only recurve the first time we evaluate a given Fibonacci number.

In all other cases we just look up the dictionary.

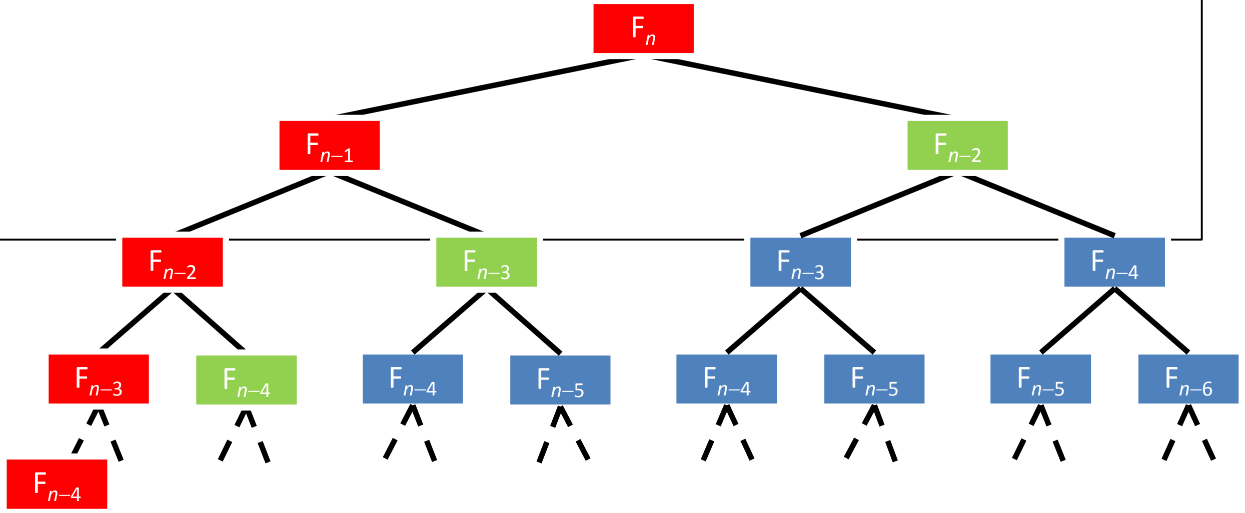
Our evaluation tree…

…becomes:

Evaluate;

Memoize;

Ignore.



Analysis

With this, Dynamic Programming, approach:

- We compute F_k once for each value $1 \leq k \leq n$;
    - n calls;
    - O(1) per call;
- We look up F_k once for each value $1 \leq k \leq n-1$;
    - $n-1$ calls;
    - O(1) per call.

So, **`fibDP`** takes $O(n)$ time to compute F_n.

In General

We can state the general technique for dynamic programming as follows:

> Solve any sub-problem once and memorize (remember) these solutions for later re-use.

Memorization is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

In General

In essence: DP is recursion + memorization.
The critical problem in using DP is the identification of the sub-problems.
The solution time for dynamic programming is derived as follows:

Multiply the number of distinct sub-problems by the solution time per sub-problem;
Note: we only solve a sub-problem once.

Turning DP on its Head

Another way to think about dynamic programming is to look at it as a *bottom up* solution.
In contrast, recursion is a *top down* solution.
We can write a bottom up Fibonacci algorithm as follows:

Bottom up Fibonacci Numbers

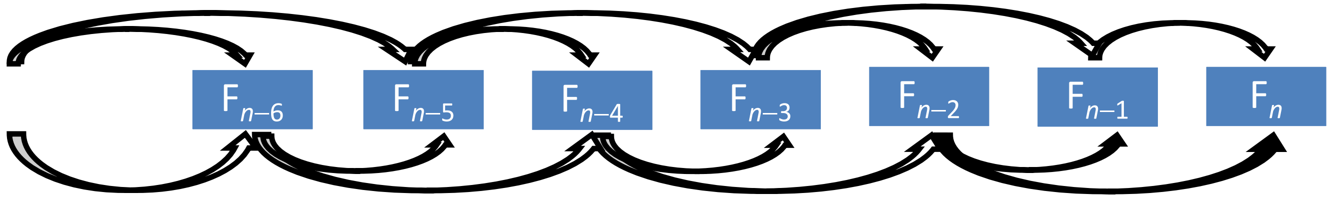
```
Procedure fibUp(n: integer): integer
      fib: dictionary = {}

      k=1
      repeat
            if k ≤ 2 then
                  f = 1
            else
                  f = fib[k-1]+fib[k-2]
            fi
            fib[k] = f
            k++
      until k==n
      return fib[n]
End procedure fibUp
```

Note that this solution completely eliminates the need for recursion in calculating the n^{th} Fibonacci number.

All dynamic programming algorithms can be transformed in this way.

Bottom Up in General

The bottom up approach to DP still involves solving the same set of sub-problems as in the top down approach. What changes is the order in which we solve them. The bottom up order can be considered as a topological sort of the problem's dependency graph.
For the Fibonacci numbers…



… so the sort order is F_1, F_2, F_3,… F_{n-3}, F_{n-2}, F_{n-1}, F_n.

Saving Space with DP

Often, the bottom up version of dynamic programming allows us to save space (memory) as well as time.
As we presented the algorithm, it used a dictionary containing *n* entries.
In fact, we only ever need the last two values; we can forget the earlier ones.
This allows us to re-write the algorithm without explicit memorization.

Memo-free Fibonacci Numbers

```
Procedure fibSmall(n: integer):integer
     prev:integer = 0
     f: integer = 1

     k=2
     repeat
          f = f+prev
          prev = f–prev
          k++
     until k == n
     return f
end procedure fibSmall
```

Dynamic Programming Example II: Shortest Paths

Let us apply the insights we have gained on dynamic programming to a second problem.:

Single source, all destinations shortest path.

We will proceed as follows:

1. Create a top down, recursive, naïve algorithm;
2. Memoize it;
3. Reconstruct it as a bottom up algorithm.

This is a useful general approach to algorithm design in dynamic programming.

Naïve algorithms are where we take a guess. . .

Step 1: the Naïve, Recursive Algorithm.

In deriving the naïve algorithm we need to introduce another key component of dynamic programming…

…guessing!

Don’t know the answer?

Guess!

Don’t just try any guess...

…try them all!

So, DP = recursion + memorization + guessing.

The best guess is the answer we are looking for.

Some Notation for Shortest Paths

Remember :

> Given a graph, $G=(V, E, W)$, find the shortest path from a starting vertex, $s \in V$, to all other vertices, $v \in V$;
> $w(u, v)$ is the weight of the edge (u, v);
> $D(s, v)$ is the length of the shortest path between s and v.

If some vertex, u, is on the shortest path from s to v then:

> $D(s, v) = D(s, u) + D(u, v)$.

Specifically, if vertex u immediately precedes vertex v in the shortest path from s to v, then:

> $D(s, v) = D(s, u) + w(u, v)$.

Our problem is that we don't know which vertex, u, to try…

> …so we guess—try them all and pick the best.

The Naïve Algorithm

```
Procedure short(V{}: vertex, E{}: edge, W():
weight, s: vertex, v: vertex)
    if v==s then     d=0
    else d = ∞
     for each u where (u,v) ∈ E
       d = min(d,short(V,E,W,s,u)+w(u,v))
     rof
    fi
    return d
End procedure short
```

This is a really bad algorithm: We compute the shortest path between s and every other vertex repeatedly. It is really easy to improve, however; Memorize the computation.

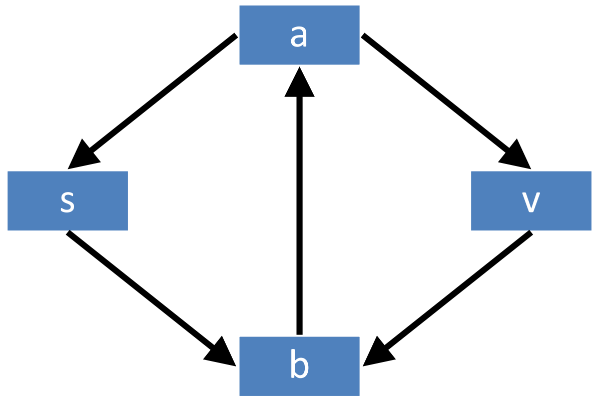
Step 2: The Memorized Algorithm

```
D: dictionary {}

Procedure shortDP(V{}: vertex,E{}: edge,W():weight, s:vertex, v:
vertex)
  if v==s then d=0
  else d = ∞
      for each u where (u,v) ∈ E
         if (u in D) then d=min(d, D[u]+w(u,v)
         else d = min(d, shortDP(V, E, W), s, u) + w(u,v))
         fi
      rof
   fi
   D[v]=d
   return d
End procedure shortDP
```

Some Analysis

Consider the following graph:
To find the shortest path $D(s, v)$ we proceed as follows:

$$\begin{tikzcd}
& D(s, v) \arrow[d] & \\
& D(s, a) \arrow[d] & \\
& D(s, b) \arrow[dl] \arrow[dr] & \\
D(s, s) & & D(s, v)
\end{tikzcd}$$



We now have a problem…

…to find $D(s, v)$ we need to evaluate $D(s, v)$.

Oops

Our “improved” algorithm has a problem.
It takes infinite time if G has one or more cycles.
If g is acyclic the algorithm is O(|V|+|E|)
We should have anticipated this…

…remember the bottom up formulation.

The order of evaluation of sub-problems is a topological sort of the dependency graph.
You can only perform a topological sort on a DAG…

…no cycles allowed.

Decycling a Graph

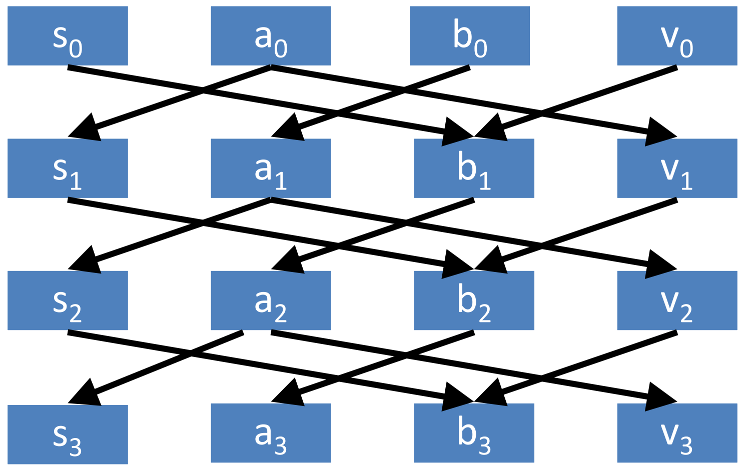
Is there some way to remove cycles from a graph? Yes, provided none of them are negative cost cycles. We replicate the graph $|V|$ times and construct a new graph as follows:

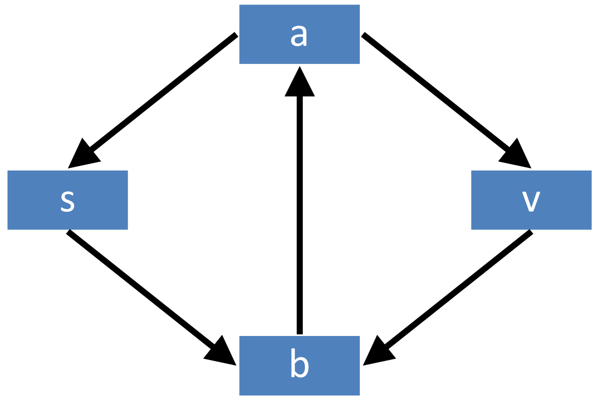
Eliminate all edges between vertices in the same copy:

If $(u, v) \in E$ in the original graph connect u_i to v_{i+1} in the new graph.

This is best seen with an example.

- Let us use our previous graph:
- This becomes:





- This new graph has $|V|^2$ vertices and $|V|\times|E|$ edges…
  - …but it has no cycles.
- We now define $D_k(s, v)$ as the shortest path from s to v that traverses exactly k edges.
- The shortest path is now the smallest of the $D_k(s, v)$ values.

So What?

We now observe that:

$D_k(\underline{s}, v) = \min (D_{k-1}(s, u)+w(u, v))$.

So, if we use our memorized DP shortest path solution algorithm on this graph we can solve our original problem, even though our graph has cycles.

The bottom up version of this $O(|V|\times|E|)$ algorithm is exactly the same as the Bellman-Ford algorithm we saw last week.

In fact, this is how the Bellman-Ford algorithm was discovered.

Crazy Eights

We are now going to examine a problem involving playing cards.

Given a sequence of playing cards find the longest valid subsequence of cards in which each card selected is “like” its neighbors.

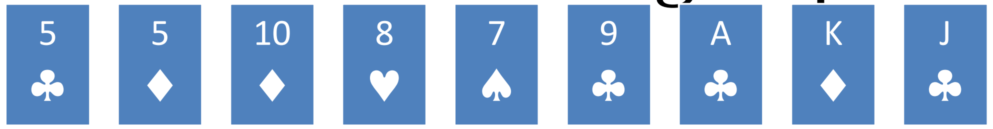
Cards are like each other if:

- Either they are of the same value;
- Or they are of the same suit;
- Or at least one of the cards is an 8.

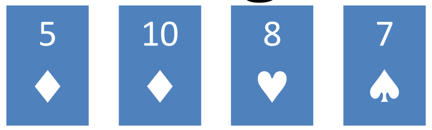
Sub sequences must be in the same left-to-right order as the original sequence but are allowed to skip cards.

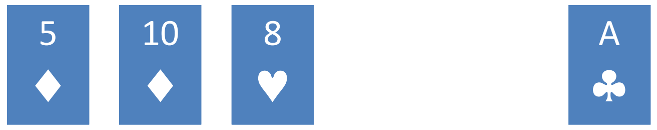
An Example

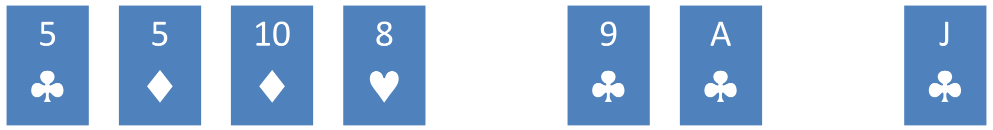
Consider the following sequence of cards:



The following are some possible subsequences







In this example the last example is the longest possible.

How to Solve it.

How can we solve this puzzle?
Surprise!
We use Dynamic Programming.
The first (and biggest) part of this process is how do we re-state the problem in a way that we can use DP to solve.
We need a directed, acyclic graph of dependencies.
Once we have that, the rest is easy.
So, how do we turn the problem into a graph?

Cards to Graphs

To convert the problem into a graph, we need to identify:

- The vertices;
- The edges;
- The weights.

We also need to formulate the problem in terms of minimizing (or, possibly, maximizing) some function of the weights.

- We call this the *objective* function.

The first of these conversion issues is easy to address:

- What are the vertices?
  - The cards.
  - Actually, we will add one more “dummy” vertex *s*.

What are the Edges?

We now define directed edges (*u*, *v*) as follows.

Edge (*u*, *v*) exists as long as:

- Card *u* lies to the left of card *v* in the original sequence.
- Card *u* is like card *v*.

Remember, any 8 is like every card.

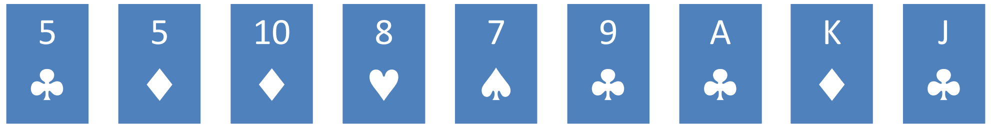
So is our starting card *s*.

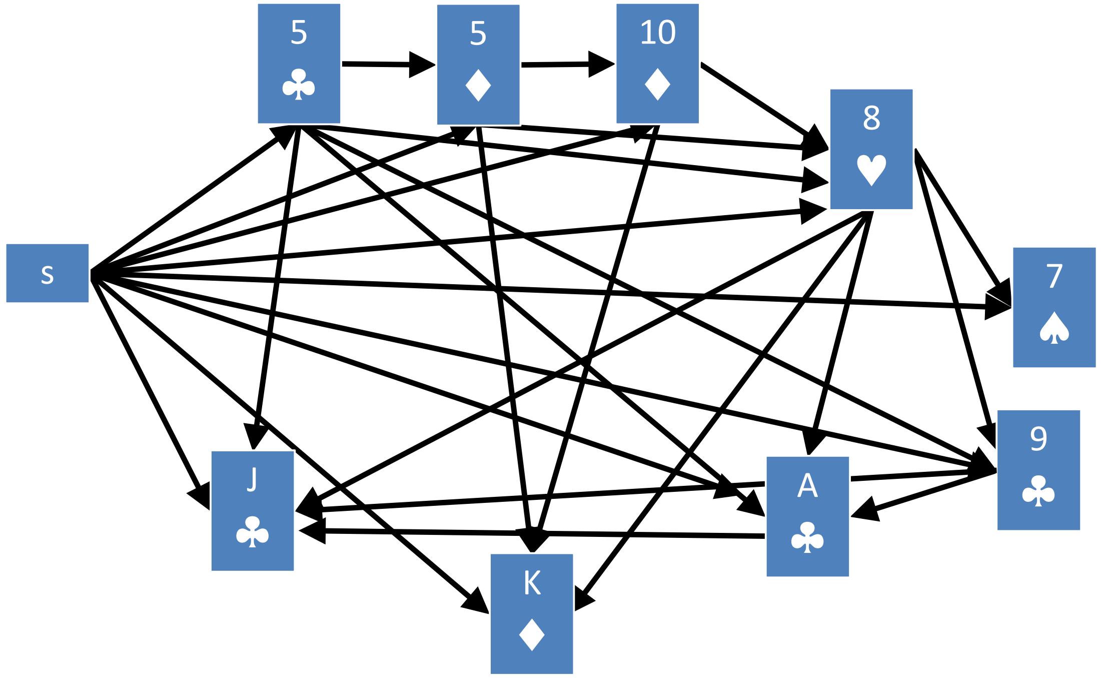
Edge Weights?

As we are only concerned with the number of cards we do not need to differentiate one edge from another.
Thus, we can assign a constant weight of one to each edge.
Our example problem is converted into the following directed graph:

The Graph

Becomes:





The Objective Function

For this problem we want to find the longest valid sub-sequence.
We note that the length of a valid sub-sequence, ending at card v in the original problem, is equal to the maximum path-length, M(s, v), from s to v in the graph we have defined.
This is defined as the sum of weights (each equal to one) of the edges in the path.
Our objective function is M(s, v) and the problem becomes:

Find $v \in V$ such that M(s, v) is maximized.

Solving the Problem

We note that:

If the maximum path from s to v passes through some vertex u then:

$M(s, v) = M(s, u) + M(u, v)$.

If u is the last vertex (card) before v then:

$M(s, v) = M(s, u) + w(u, v) = M(s, u) + 1$.

So our problem becomes:

```
For each $v$ in $V$:
    Find $M(s, v)$
Find the maximum of all the M values.
```

Crazy Eights using Dynamic Programming

The solution for vertex *v* becomes:

```
M: dictionary = {}
Procedure crazyDP(V{}: vertex, E{}: edge, s: vertex, v: vertex)
   for each v in V
      m=0
      if v ≠ s then
         for each u where (u,v) ∈ E
            if (u in M) then  m = max (m, D[u]+1)
            else  m = max(m, crazyDP(V, E, s, u) + 1)
            fi
         rof
      fi
      M[v] = m
  rof
End procedure crazyDP
```

Driver Program

We now need a driver program to iterate over all the vertices.

```
for each v in V
        crazyDP(V, E, s, v)
rof
solution = max M[v]
```

We can use the standard methods to optimize the algorithm:

E.g. make M a max heap ordered on path length;

Track the preceding card in each longest sequence:

Another Approach:

Consider the following, alternate formulation of the problem:

> *V* is still the cards plus *s*, a dummy starting card;
>
> *E* is the same as we defined before;
>
> *W* is set to –1 for each edge in *E*.

Now we have a simple shortest path problem. We could use Bellman-Ford to solve this.

Is This Good?

Bellman-Ford is O(|V|x|E|)

If we have N cards.

- |V| $\in$ O(N);
- |E| $\in$ O(N^2)

So the overall running time is O(N^3)

We can do better than this:

Better than N^3

We note that the graph associated with the problem is acyclic:

All edges go from left to right.

This means that we can find a shortest path solution in $O(|V|+|E|) = O(N+N^2) = O(N^2)$

We just need a good order in which to consider the vertices…

…we need a topological sort.

How do we get one?

Order Zero Topological Sort

We can get a topological sort of our problem graph in O(0) time.
Yes—no time at all!
How?
We already have it.
It is the original order of our sequence of cards!
We can now write a non-recursive, bottom up, DP procedure that encapsulates the insights we have gained.

Let C be the original sequence of cards:

$C=(c_0, c_1, c_2, c_3, \ldots, c_{N-1}, c_N)$;

Note: $c_0 = s$.

We can write a bottom up algorithm as follows:

Bottom Up Crazy Eights

```
Procedure crazyUp(C(): cards)
     Length: dictionary = {}
     Length[0] = 0  and  maxLength=0
     for i in 1..N
          len = 1
          for j in i-1..0
               if C(i) is like C(j) then
                    len = max(len, Length(j)+1)
               fi
          rof
          Length(i) = len
          maxLength = max (maxLength, len)
     rof
     return maxLength
End procedure crazyUp
```

Notes

We have completely eliminated the graph;

This is not strictly true…

…we have used an *implicit* representation of the graph.

Our test:

`if C(i) is like C(j)`;

Is equivalent to traversing the edges on the dependency graph.

The order of the cards:

`c(0), c(1), …, c(n)`;

Is the topological sort of the vertices.

The only thing that remains of the original formulation is:

The dictionary, **`Length[]`**.

DP Steps

The DP process is broken into 5 general steps (with analysis):

1. Define the sub-problems (determine number of sub-problems);
2. Guess the solution to a sub-problem (determine number of guesses);
3. Relate sub-problems via recursion (determine time per sub-problem);
4. Solve the sub-problems systematically, ensure sub-problem dependency is acyclic (#sub-problems $\times$ time per sub-problem);
   - Recurve and memorize (top down);
   - Topological sort and loop (bottom up);
5. Solve original problem by combining sub-problem solutions (O(1)).