HMKANG@UMICH.EDU

MODULE 1 / UNIT 8

DYNAMIC PROGRAMMING

Today

- Example : Fibonacci sequence
- Concept : Dynamic programming
- Example : Manhattan Tourist Problem
- Example : Edit distance problem

A recursive version of Fibonacci numbers

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & n > 1 \\ 1 & n = 1 \\ 0 & n = 0 \end{cases}$$

- Define a function `fib(n)` that returns Fibonacci number F_n using recursions

A simple python implementation

```
def fib(n):
    if ( n < 2 ):
        return n
    else:
        return fib(n-1)+fib(n-2)
```

Correctness of the function

```
print(fib(2))
print(fib(3))
print(fib(4))
print(fib(5))
print(fib(6))
print(fib(7))
print(fib(8))
print(fib(9))
print(fib(10))
print(fib(20))
```

```
1
2
3
5
8
13
21
34
55
6765
```

How about the computational efficiency?

```
%%time
fib(20)
```

```
Wall time: 5.01 ms

6765
```

```
%%time
fib(30)
```

```
Wall time: 516 ms

832040
```

```
%%time
fib(35)
```

```
Wall time: 6.59 s

9227465
```

```
%%time
fib(40)
```

```
Wall time: 55.3 s

102334155
```

How about the computational efficiency?

```
%%time
fib(20)
```

```
Wall time: 5.01 ms

6765
```

```
%%time
fib(35)
```

```
Wall time: 6.59 s

9227465
```

```
%%time
fib(30)
```

```
Wall time: 516 ms

832040
```

```
%%time
fib(40)
```

```
Wall time: 55.3 s

102334155
```

Why takes so long?

A loopy implementation of Fibonacci numbers

```
def fib_loopy(n):
    s1 = 0
    s2 = 1
    for i in range(2,n+1):
        (s2,s1) = (s2+s1,s2)
    return(s2)
```

.. is much faster.. why?

```
%%time
print(fib_loopy(20))
print(fib_loopy(30))
print(fib_loopy(35))
print(fib_loopy(40))
```

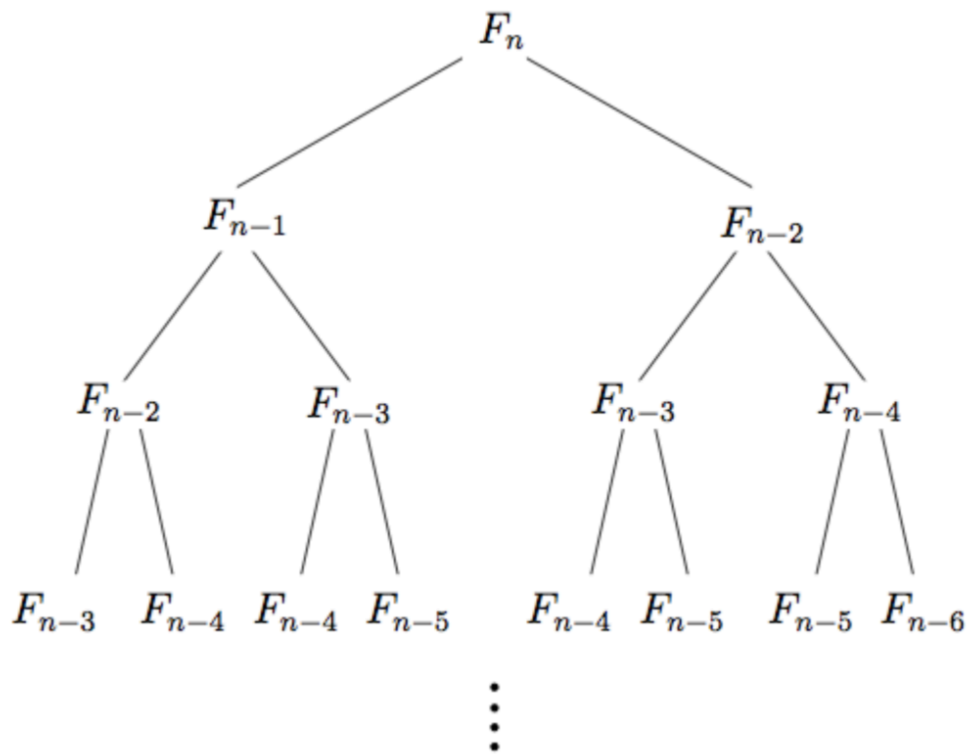
```
6765
832040
9227465
102334155
CPU times: user 279 µs, sys: 259 µs, total: 538 µs
Wall time: 305 µs
```

Why was recursive version so slow?

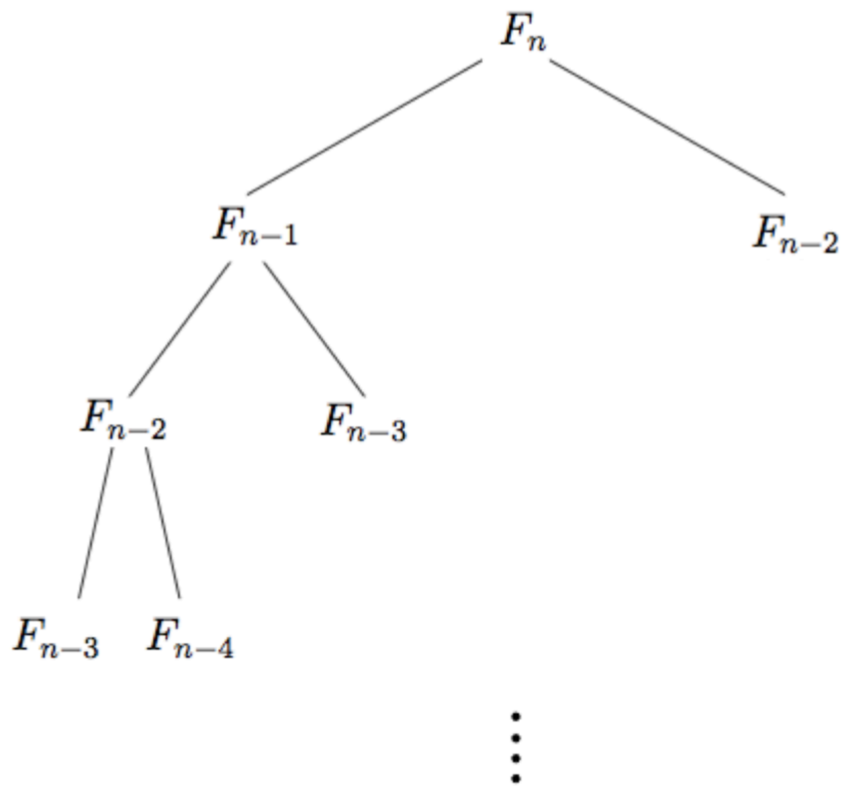
```python
def fib(n):
    if ( n < 2 ):
        return n
    else:
        return fib(n-1)+fib(n-2)
```

```python
def fib_loopy(n):
    s1 = 0
    s2 = 1
    for i in range(2,n+1):
        (s2,s1) = (s2+s1,s2)
    return(s2)
```

Recursive function calls are redundant



A reasonable alternative:



How?

Dynamic Programming

- **Problems can divide into subproblems**

- **Subproblems are overlapping**
  - For example, subproblems share subsubproblems
  - Key difference from divide-and-conquer algorithm

- **Strategy**
  - Solve each subproblem only once
  - Store the answer of the subproblem for later use

DP implemention

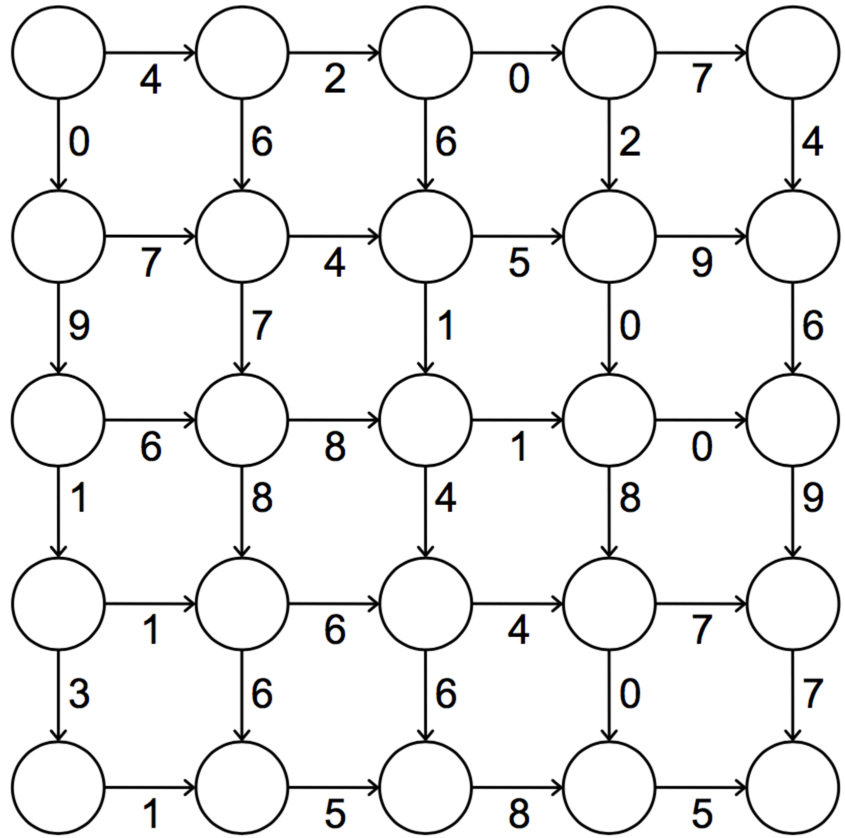
```
def fib_dp(n, stored): ## assume that empty list will be provided
    if ( len(stored) == 0 ):
        stored.extend((n+1) * [None])
    if ( stored[n] is None ):
        if ( n < 2 ):
            stored[n] = n
        else:
            stored[n] = fib_dp(n-1,stored) + fib_dp(n-2,stored)
    return(stored[n])
```

Results with DP implementation

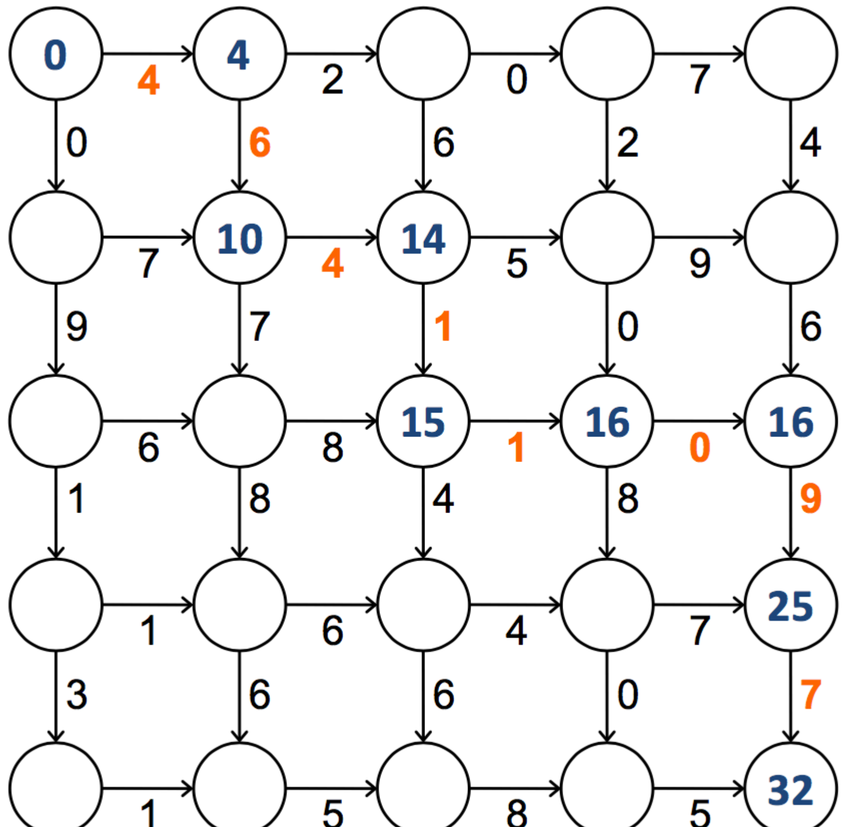
```
%%time
print(fib_dp(20,[]))
print(fib_dp(30,[]))
print(fib_dp(35,[]))
print(fib_dp(40,[]))
```

```
6765
832040
9227465
102334155
CPU times: user 801 µs, sys: 702 µs, total: 1.5 ms
Wall time: 897 µs
```

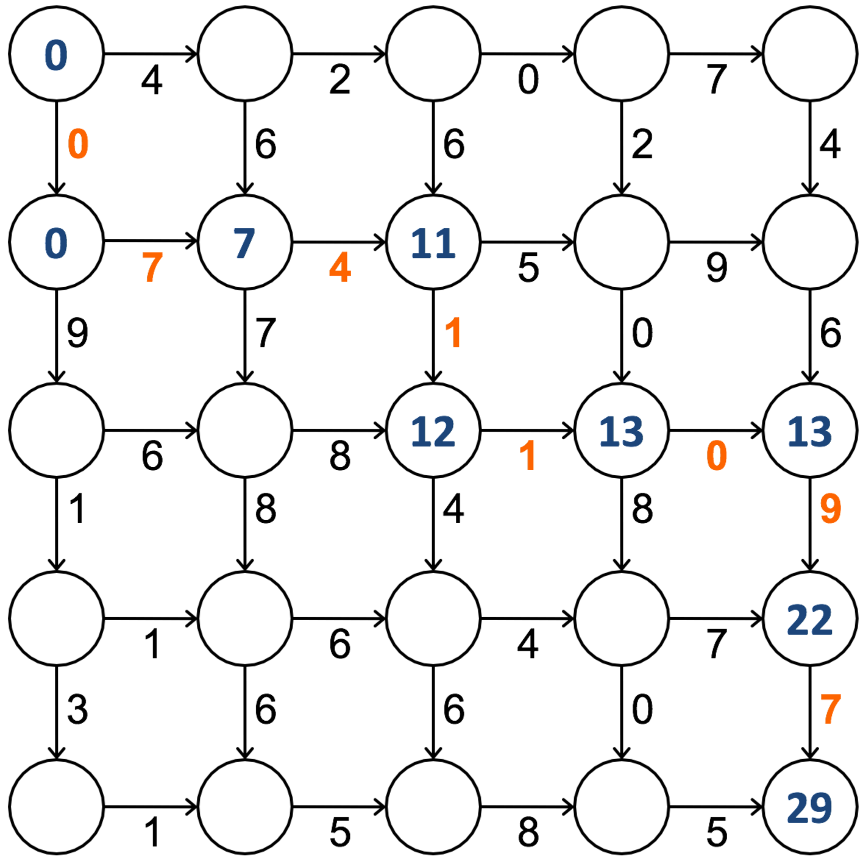
The Manhattan Tourist Problem (MTP)



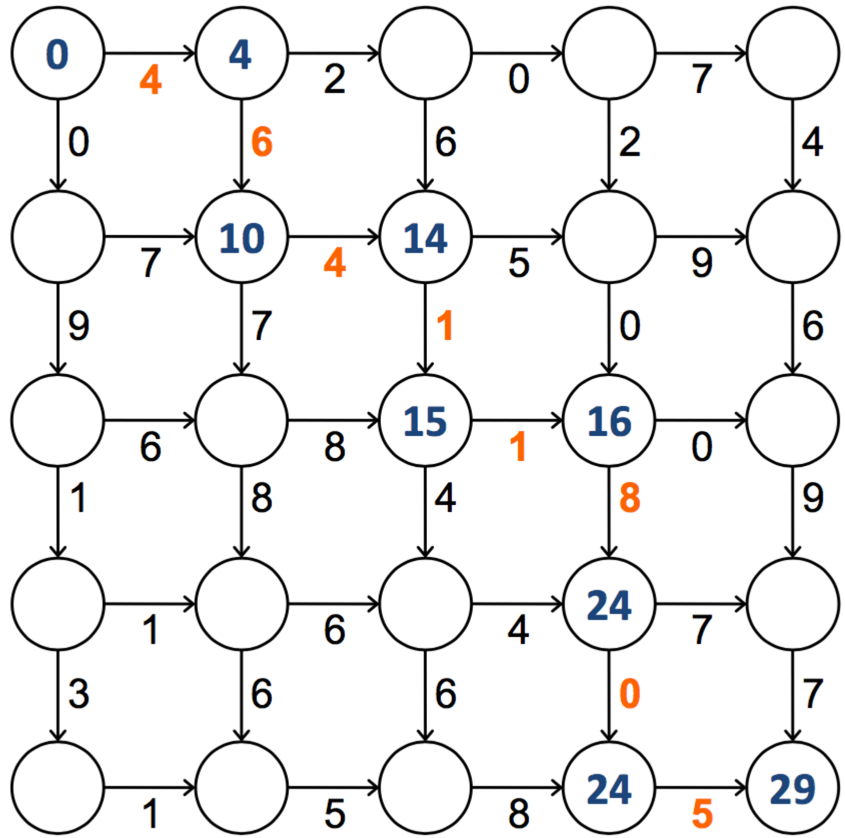
A possible solution



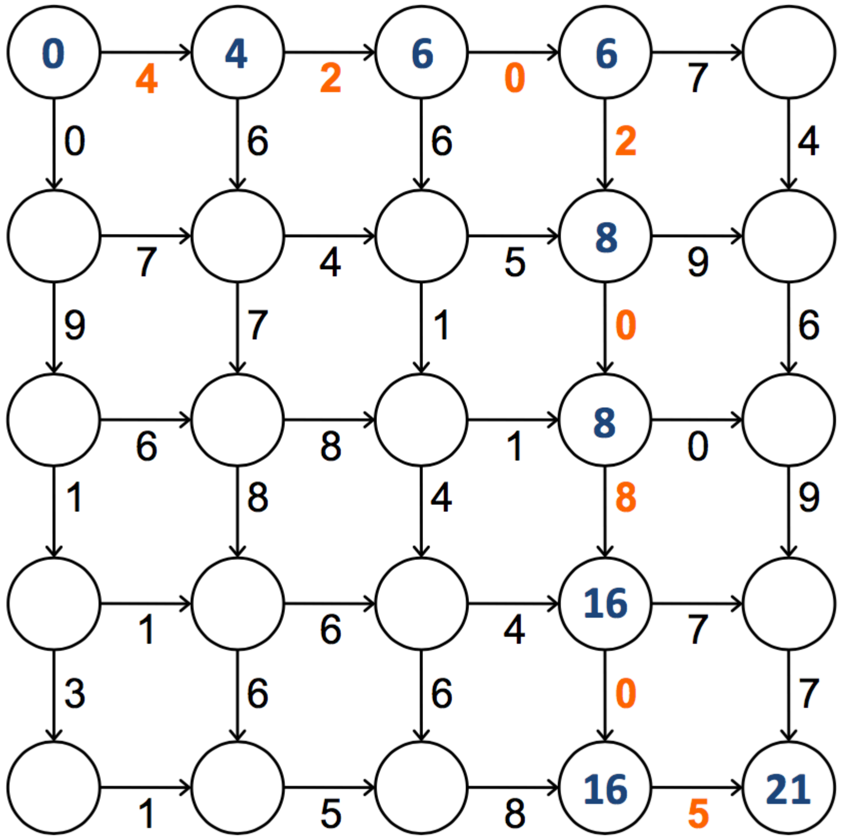
A greedy solution – is this optimal?



Another possible solution



Is this an optimal solution?



A brute-force algorithm

1. Enumerate all possible paths
2. Calculate the cost of each possible path
3. Pick the path that produces a minimum cost

What is the number of all possible paths for (r x c) problem?

Problem setup

- **Calculating the minimal cost**
  - What is the minimum cost of optimal path?

- **Obtaining the optimal path**
  - Can we obtain an optimal path that achieves the optimal cost?

Calculating the minimal cost

- How can the problem be divided into subproblems?
- Do subproblems overlap? How?
- What information should be stored to avoid redundancy?

Dynamic Programming Idea

1. **Let C(r,c) is the optimal cost from (0,0) to (r,c)**

2. **Let's pretend that we know all optimal solutions before C(r,c)**
   - C(i,j) is known for all $0 \leqslant i \leqslant r$ and $0 \leqslant j \leqslant c$ except for (i,j)=(r,c)

3. **Then C(r,c) must be one of the two**
   1. C(r-1,c) + cost from (r-1,c) to (r,c)
   2. C(r,c-1) + cost from (r,c-1) to (r,c)

   And smaller value is C(r,c)

Formulating the Idea

$$C(r,c) : \text{optimal cost from}(0,0)\ \mathit{to}\ (r,c)$$

$$v(r,c) : \text{cost from}(r,c) \text{ to } (r+1,c)$$

$$h(r,c) : \text{cost from}(r,c) \text{ to } (r,c+1)$$

$$C(r,c) = \quad \min \begin{cases} C(r-1,c) + v(r-1,c) \\ C(r,c-1) + h(r,c-1) \end{cases}$$

Is this sufficient?

Adding boundary conditions

$$C(r,c) : \text{optimal cost from}(0,0)\ to\ (r,c)$$

$$v(r,c) : \text{cost from}(r,c) \text{ to } (r+1,c)$$

$$h(r,c) : \text{cost from}(r,c) \text{ to } (r,c+1)$$

$$C(r,c) = \begin{cases} 0 & r=0, c=0 \\ C(r,c-1)+h(r,c-1) & r=0, c>0 \\ C(r-1,c)+v(r-1,c) & r>0, c=0 \\ \min\begin{cases} C(r-1,c)+v(r-1,c) \\ C(r,c-1)+h(r,c-1) \end{cases} & r>0, c>0 \end{cases}$$

Implementation in `Rcpp`

- **Defining a C++ function**
  - Input parameters
    - `NumericMatrix H` : R x (C-1) cost matrix
    - `NumericMatrix V` : (R-1) x C cost matrix
    - `int r, c` : index of horizontal and vertical coordinates
    - *(and we need one more thing...)*
  - Return value
    - Optimal cost to reach to the (r,c) coordinate

Dynamic programming part of MTP

```
// We assume that C contains all negative values initially
double mtp_dp(NumericMatrix& H, NumericMatrix& V, int r, int c, NumericMatrix& C) {
  if ( C(r,c) < 0 ) { // need to compute and store
    if ( r == 0 ) {
      if ( c == 0 ) C(r,c) = 0;
      else C(r,c) = mtp_dp(H,V,r,c-1,C) + H(r,c-1);
    }
    else {
      if ( c == 0 ) C(r,c) = mtp_dp(H,V,r-1,c,C) + V(r-1,c);
      else {
        double hcost = mtp_dp(H,V,r,c-1,C) + H(r,c-1);
        double vcost = mtp_dp(H,V,r-1,c,C) + V(r-1,c);
        C(r,c) = hcost > vcost ? vcost : hcost;
      }
    }
  }
  return C(r,c);
}
```

Making R/C++ interface

```
// [[Rcpp::export]]
double mtp(NumericMatrix H, NumericMatrix V) {
  int r = H.nrow();
  int c = H.ncol() + 1;
  if ( ( V.nrow() + 1 != r ) || ( V.ncol() != c ) )
    stop("The dimensions of H and V do not match");
  NumericMatrix C(r, c);
  fill(C.begin(), C.end(), -1.0);

  return mtp_dp(H, V, r-1, c-1, C);
}
```

Running from R

```
hcost <- matrix(c(4,7,6,1,1,2,4,8,6,5,0,5,1,4,8,7,9,0,7,5),5,4)
vcost <- matrix(c(0,9,1,3,6,7,8,6,6,1,4,6,2,0,8,0,4,6,9,7),4,5)
```

```
print(hcost)
```

```
     [,1] [,2] [,3] [,4]
[1,]    4    2    0    7
[2,]    7    4    5    9
[3,]    6    8    1    0
[4,]    1    6    4    7
[5,]    1    5    8    5
```

```
print(vcost)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    6    6    2    4
[2,]    9    7    1    0    6
[3,]    1    8    4    8    9
[4,]    3    6    6    0    7
```

```
mtp(hcost,vcost)
```

```
[1] 21
```

Time complexity of the dynamic programming

How many times was the function called?

- Each C(r,c) is evaluated at most once
- Every node before (r,c) will be evaluate at least once
- The total time complexity is O(rc)

Reconstructing the optimal path

- Does the problem become different from the previous one? How?

- What information should be stored to avoid redundancy?

- How can the stored information be used to reconstruct the optimal path?

How can we reconstruct the optimal path?

- **Knowing optimal cost is useful, but..**
  - Knowing the path leading to optimal cost is also important.

- **More information is needed to reconstruct the optimal path.**

- **The output we want is something like..**

```
Optimal cost is 21
Optimal path is (0,0)--[4]-->(0,1)--[2]-->(0,2)
--[0]-->(0,3)--[2]-->(1,3)--[0]-->(2,3)--[8]-->
(3,3)--[0]-->(4,3)--[5]-->(4,4)
```

The Idea : Backtracking

1. When recording optimal path into a junction, record ‘from which path’ information as well.

2. By backtracking from the destination to the source, one can reconstruct the optimal path

```cpp
double mtp2_dp(NumericMatrix& H, NumericMatrix& V, int r, int c, NumericMatrix& C,
 LogicalMatrix& P) {
  if ( C(r,c) < 0 ) { // need to compute and store
    if ( r == 0 ) {
      if ( c == 0 ) C(r,c) = 0;
      else {
        C(r,c) = mtp2_dp(H,V,r,c-1,C,P) + H(r,c-1);
        P(r,c) = true;
      }
    }
    else {
      if ( c == 0 ) {
        C(r,c) = mtp2_dp(H,V,r-1,c,C,P) + V(r-1,c);
        P(r,c) = false;
      }
      else {
        double hcost = mtp2_dp(H,V,r,c-1,C,P) + H(r,c-1);
        double vcost = mtp2_dp(H,V,r-1,c,C,P) + V(r-1,c);
        C(r,c) = hcost > vcost ? vcost : hcost;
        P(r,c) = hcost < vcost;
      }
    }
  }
  return C(r,c);
}
```

Reconstructing an optimal path

```
string mtp2_optpath(int32_t r, int32_t c, NumericMatrix& H, NumericMatrix& V,
 LogicalMatrix& P) {
  std::string path = "(" + to_string(r) + "," + to_string(c) + ")";
  while ( ( r > 0 ) || ( c > 0 ) ) {
    int w;
    if ( P(r,c) ) { w = H(r,c-1); --c; }
    else { w = V(r-1,c); --r; }
    path = string("(") + to_string(r) + "," + to_string(c) + ")--[" + to_strin
g(w) + "]-->" + path;
  }
  return path;
}
```

Function interfacing with R

```
// [[Rcpp::export]]
List mtp2(NumericMatrix H, NumericMatrix V) {
  int r = H.nrow();
  int c = H.ncol() + 1;
  if ( ( V.nrow() + 1 != r ) || ( V.ncol() != c ) )
    stop("The dimensions of H and V do not match");
  NumericMatrix C(r, c);
  LogicalMatrix P(r, c);
  fill(C.begin(), C.end(), -1.0);

  double optcost = mtp2_dp(H, V, r-1, c-1, C, P);
  string optpath = mtp2_optpath(r-1, c-1, H, V, P);
  return List::create(Named("cost")=optcost,
                      Named("path")=optpath);
}
```

An example result

```r
hcost <- matrix(c(4,7,6,1,1,2,4,8,6,5,0,5,1,4,8,7,9,0,7,5),5,4)
vcost <- matrix(c(0,9,1,3,6,7,8,6,6,1,4,6,2,0,8,0,4,6,9,7),4,5)
mtp2(hcost,vcost)
```

```
$cost
[1] 21

$path
[1] "(0,0)--[4]-->(0,1)--[2]-->(0,2)--[0]-->(0,3)--[2]-->(1,3)--
[0]-->(2,3)--[8]-->(3,3)--[0]-->(4,3)--[5]-->(4,4)"
```

Dynamic programming : A smart recursion

- **Dynamic programming is recursion without repetition**
  - Formulate the problem recursively
  - Build the solution bottom up
  - The number of function evaluations is constant for each parameter.
- **Dynamic programming is an expansion of divide and conquer**
  - There are problems that cannot be partitioned into two independent subproblems.
- **Dynamic programming requires additional cost**
  - Storage space to save the answer to subproblems.
  - If the answers are long, then it could consume lots of memory

Edit distance problem

- **Edit distance**
  - Minimum number of insertions, deletions, and substitutions required to transform one word into another

- **An example : EditDistance(FOOD,MONEY) = 4**

  FOOD

  MOOD

  MOND

  MONED

  MONEY

More examples of edit distance

			F	O	O		D			
			M	O	N	E	Y			
A	L	G	O	R		I		T	H	M
A	L		T	R	U	I	S	T	I	C

Brainstorm: How to compute edit distance?

- **Use dynamic programming!**

- **Function** `EditDistance(s1, s2, i, j):`

  1. Computes optimal edit distance between length `i` prefix of `s1` and length `j` prefix of `s2`

  2. Construct a recursion to calculate the value in a divide-and-conquer fashion.

  3. Store the optimal cost once computed to avoid redundancy

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

Formulating a recursion

$$D(i,j) : \text{edit distance between } s_1[1..i] \; and \; s_2[1..j]$$

$$D(i,j) = \quad \min \begin{cases} D(i-1, j-1) + I(s_1[i] \neq s_2[j]) \\ D(i-i, j) + 1 \\ D(i, j-1) + 1 \end{cases}$$

- Note that indices are 1-based in this formula
- Does this look like a right idea?
- Is the formulation complete?

Refining the recursion

$$D(i,j): \text{edit distance between } s_1[1..i] \; and \; s_2[1..j]$$

$$D(i,j) = \begin{cases} \min \begin{cases} D(i-1, j-1) + I(s_1[i] \neq s_2[j]) \\ D(i-i, j) + 1 \\ D(i, j-1) + 1 \end{cases} & i > 0, j > 0 \\ D(i-1, j) + 1 & i > 0, j = 0 \\ D(i, j-1) + 1 & i = 0, j > 0 \\ 0 & i = 0, j = 0 \end{cases}$$

```
int edist_dp(string& s1, string& s2, IntegerMatrix& cost, int r, int c)
 {
  if ( cost(r,c) < 0 ) {
    if ( r == 0 ) {
      if ( c == 0 ) cost(r,c) = 0;
      else cost(r,c) = edist_dp(s1, s2, cost, r, c-1) + 1;
    }
    else if ( c == 0 ) cost(r,c) = edist_dp(s1, s2, cost, r-1, c) + 1;
    else {
      int iDist = edist_dp(s1, s2, cost, r, c-1) + 1;
      int dDist = edist_dp(s1, s2, cost, r-1, c) + 1;
      int mDist = edist_dp(s1,s2,cost,r-1,c-1)+(s1[r-1]!=s2[c-1]);
      if ( iDist < dDist ) cost(r,c) = iDist < mDist ? iDist : mDist;
      else  cost(r,c) = dDist < mDist ? dDist : mDist;
    }
  }
  return cost(r,c);
}
```

Interfacing with R

```
// [[Rcpp::export]]
int edit_distance(string s1, string s2) {
  int r = (int)s1.size();
  int c = (int)s2.size();
  IntegerMatrix cost(r+1,c+1);
  fill(cost.begin(), cost.end(), -1.0);
  return edist_dp(s1, s2, cost, r, c);
}
```

Running examples

```
edit_distance("FOOD","MONEY")
```

```
[1] 4
```

```
edit_distance("ALGORITHM","ALTRUISTIC")
```

```
[1] 6
```

Summary

- **Dynamic programming : A smart recursion**
- **Manhattan tourist problem**
  - Calculating optimal cost
  - Backtracking an optimal path
- **Edit distance problem**