

Biostatistics 615/815 Lecture 8: Dynamic Programming

Hyun Min Kang

September 27th, 2012

Removing an element from a list

```
myList.h
template <class T>
bool myList<T>::remove(const T& x) {
    if ( head == NULL )
        return false;    // NOT_FOUND if the list is empty
    else {
        // call head->remove will return the object to be removed
        myListNode<T>* p = head->remove(x, head);
        if ( p == NULL ) { // if NOT_FOUND return false
            return false;
        }
        else {           // if FOUND, delete the object before returning true
            delete p;
            return true;
        }
    }
}
```

Removing an element from a list

```
myListNode.h
template <class T>
// pass the pointer to [prevElement->next] so that we can change it
myListNode<T>* myListNode<T>::remove(const T& x, myListNode<T>*& prevNext) {
    if ( value == x ) { // if FOUND
        prevNext = next; // *pPrevNext was this, but change to next
        next = NULL; // disconnect the current object from the list
        return this; // and return it so that it can be destroyed
    }
    else if ( next == NULL ) {
        return NULL; // return NULL if NOT_FOUND
    }
    else {
        return next->remove(x, next); // recursively call on the next element
    }
}
```

Key algorithms

```
REMOVE(node, x)
1 If node.key == x
    1 If the node is leaf, remove the node
    2 If the node only has left child, replace the current node to the left child
    3 If the node only has right child, replace the current node to the right child
    4 Otherwise, pick either maximum among left sub-tree or minimum among right subtree and substitute the node into the current node
2 If x < node.key
    1 Call REMOVE(node.left, x) if node.left exists
    2 Otherwise, return NOTFOUND
3 If x > node.key
    1 Call REMOVE(node.right, x) if node.right exists
    2 Otherwise, return NOTFOUND
```

Binary search tree : REMOVE

myTree.h

```
template <class T>
myTreeNode<T>* myTreeNode<T>::remove(const T& x, myTreeNode<T>* pSelf) {
    if ( x == value ) { // key was found
        if ( ( left == NULL ) && ( right == NULL ) ) { // no child
            pSelf = NULL;
            return this;
        }
        else if ( left == NULL ) { // only left is NULL
            pSelf = right;
            right = NULL;
            return this;
        }
        else if ( right == NULL ) { // only right is NULL
            pSelf = left;
            left = NULL;
            return this;
        }
        // ....
    }
}
```

Binary search tree : REMOVE

myTreeNode.h

```
else { // neither left nor right is NULL
    // choose which subtree to delete
    myTreeNode<T>* p;
    const T& l = left->getMax();
    const T& r = right->getMin();
    if ( value - l < r - value ) { // replace with closer value
        p = left->remove(l, left);
        value = l;
    }
    else {
        p = right->remove(r, right);
        value = r;
    }
    return p;
}
}
```

Binary search tree : REMOVE

myTreeNode.h

```
else if ( x < value ) {
    if ( left == NULL )
        return NULL;
    else
        return left->remove(x, left);
}
else { // x > value
    if ( right == NULL )
        return NULL;
    else
        return right->remove(x, right);
}
}
```

Binary search tree : GETMAX and GETMIN

myTreeNode.h

```
template <class T>
const T& myTreeNode<T>::getMax() { // return the largest value
    if ( right == NULL ) return value;
    else return right->getMax();
}

template <class T>
const T& myTreeNode<T>::getMin() { // return the smallest value
    if ( left == NULL ) return value;
    else return left->getMin();
}
```

If you want to print a tree...

```
myTreeNode.h
template <class T> void myTreeNode<T>::print() {
    std::cout << "[ ";
    if ( left != NULL ) left->print();
    else std::cout << "NIL";
    std::cout << " , (" << value << " , " << size << " ) , ";
    if ( right != NULL ) right->print();
    else std::cout << "NIL";
    std::cout << " ]";
}
```

```
myTree.h
template <class T> void myTree<T>::print() {
    if ( pRoot != NULL ) pRoot->print();
    else std::cout << "(EMPTY TREE)";
    std::cout << std::endl;
}
```

Summary - Binary Search Tree

- Key Features
 - Fast insertion, search, and removal
 - Implementation is much more complicated
- Class Structure
 - myTree class to keep the root node
 - myTreeNode class to store key and up to two children
- Key Algorithms
 - Insert** : Traverse the tree in sorted order and create a new node in the first leaf node.
 - Search** : Divide-and-conquer algorithms
 - Remove** : Move the nearest leaf element among the subtree and destroy it.

Recap: Divide and conquer algorithms

Good examples of divide and conquer algorithms

- TOWEROFHANOI
- MERGESORT
- QUICKSORT
- BINARYSEARCHTREE algorithms

These algorithms divide a problem into smaller and disjoint subproblems until they become trivial.

A divide-and-conquer algorithms for Fibonacci numbers

Fibonacci numbers

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

A recursive implementation of fibonacci numbers

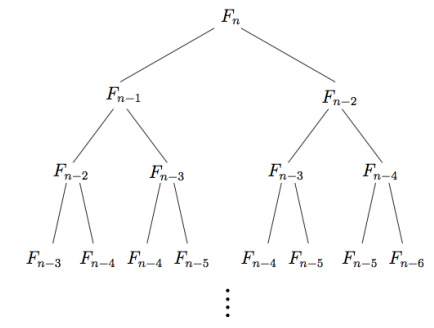
```
int fibonacci(int n) {
    if ( n < 2 ) return n;
    else return fibonacci(n-1)+fibonacci(n-2);
}
```

Performance of recursive FIBONACCI

Computational time

- 4.4 seconds for calculating F_{40}
- 49 seconds for calculating F_{45}
- ∞ seconds for calculating F_{100} !

What is happening in the recursive FIBONACCI



Time complexity of redundant FIBONACCI

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

$$T(1) = 1$$

$$T(0) = 1$$

$$T(n) = F_{n+1}$$

The time complexity is exponential

A non-redundant FIBONACCI

```
int fibonacci(int n) {
    int* fibs = new int[n+1];
    fibs[0] = 0;
    fibs[1] = 1;
    for(int i=2; i <= n; ++i) {
        fibs[i] = fibs[i-1]+fibs[i-2];
    }
    int ret = fibs[n];
    delete [] fibs;
    return ret;
}
```

Key idea in non-redundant FIBONACCI

- Each F_n will be reused to calculate F_{n+1} and F_{n+2}
- Store F_n into an array so that we don't have to recalculate it

A recursive, but non-redundant FIBONACCI

```
int fibonacci(int* fibs, int n) {
    if ( fibs[n] > 0 ) {
        return fibs[n]; // reuse stored solution if available
    }
    else if ( n < 2 ) {
        return n; // terminal condition
    }
    fibs[n] = fibonacci(n-1) + fibonacci(n-2); // store the solution once computed
    return fibs[n];
}
```

Dynamic programming

Key components of dynamic programming

- Problems that can be divided into subproblems
- Overlapping subproblems - subproblems share subsubproblems
- Solves each subsubproblem just once and then saves its answer

Why dynamic programming?

According to wikipedia... *"The word 'dynamic' was chosen because it sounded impressive, not because how the method works"*

Examples of dynamic programming

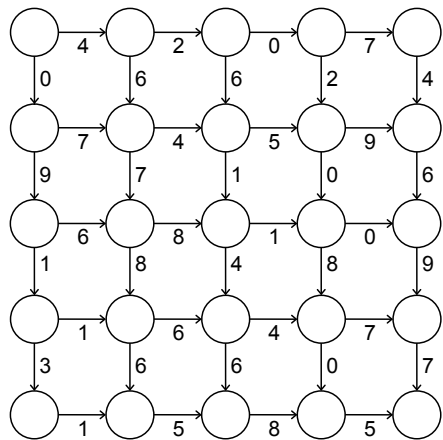
- Shortest path finding algorithms
- DNA sequence alignment
- Hidden markov models

Steps of dynamic programming

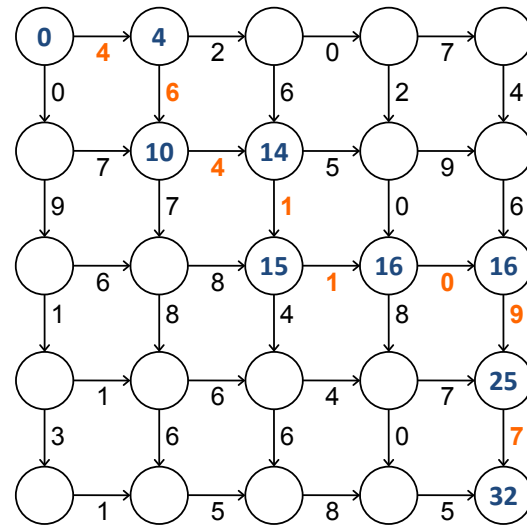
- Characterize the structure of an (optimal) solution
- Recursively define the value of an (optimal) solution
- Compute the value of an (optimal) solution, typically in a bottom-up fashion
- Construct an optimal solution from computed information.

The Manhattan tourist problem

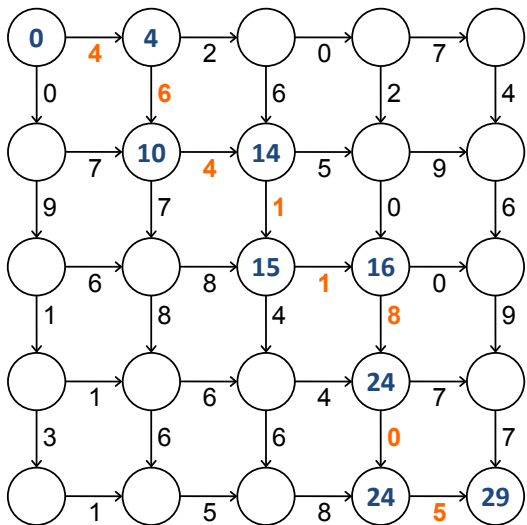
Find the cost-optimal path from left-top corner to right-bottom corner



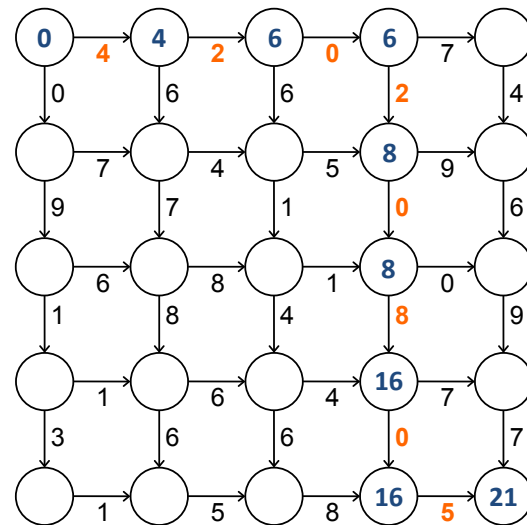
One possible (but not optimal) solution



A slightly better, but still not an optimal solution



And here comes an optimal solution



A brute-force algorithm

Algorithm BRUTEFORCEMTP

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

Time complexity

- Number of possible paths are $\binom{n_r+n_c}{n_r}$
- Super-exponential growth when n_r and n_c are similar.

A "dynamic" structure of the solution

- Let $C(r, c)$ be the optimal cost from $(0, 0)$ to (r, c)
- Let $h(r, c)$ be the weight from (r, c) to $(r, c + 1)$
- Let $v(r, c)$ be the weight from (r, c) to $(r + 1, c)$
- We can recursively define the optimal cost as

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

- Once $C(r, c)$ is evaluated, it must be stored to avoid redundant computation.

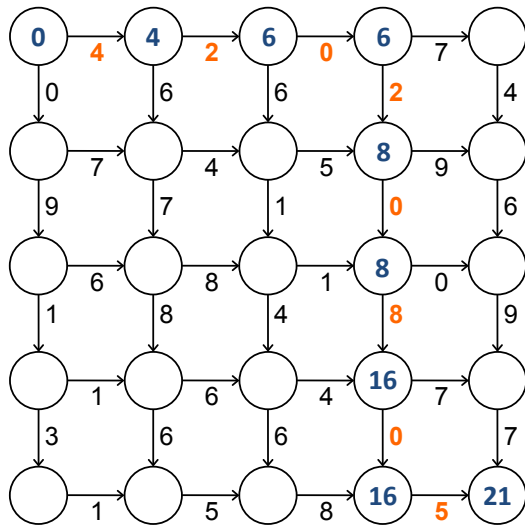
Time complexity of the "dynamic" solution

- Each recursive step takes a constant time
- Each $C(r, c)$ is evaluated at most once.
- Total time complexity is $\Theta(n_r n_c)$.
- Like Fibonacci search, the time complexity would be super exponential if $C(r, c)$ is not stored and redundantly evaluated.

Reconstructing the optimal path

- Optimal cost does not automatically produce optimal path.
- When choosing smaller-cost path between two alternatives, store the decision
- Backtrack from the destination to the source based on the stored decision

Example of backtracking the path



Implementing Manhattan tourist algorithm

```
Matrix615.h
#include <vector>

template <class T>
class Matrix615 {
public:
    std::vector< std::vector<T> > data; // vector of vector : 2D array
    Matrix615(int nrow, int ncol, T val = 0) {
        data.resize(nrow); // make n rows
        for(int i=0; i < nrow; ++i) {
            data[i].resize(ncol, val); // make n cols with default value val
        }
    }
    int rowNums() { return (int)data.size(); }
    int colNums() { return ( data.size() == 0 ) ? 0 : (int)data[0].size(); }
};
```

Manhattan tourist problem : main()

```
MTP.cpp
#include <iostream>
#include "Matrix615.h"
int main(int argc, char** argv) {
    int nrows=5, ncols=5;
    // hw stores horizontal weights, vw stores vertical weights
    Matrix615<int> hw(nrows,ncols-1), vw(nrows-1,ncols);

    hw.data[0][0] = 4; hw.data[0][1] = 2; ...
    vw.data[0][0] = 0; vw.data[0][1] = 6; ...

    Matrix615<int> cost(nrows,ncols), move(nrows,ncols);
    // calculate the optimal cost, recording the backtracking info
    int optCost = optimalCost(hw,vw,cost,move,nrows-1,ncols-1);
    std::cout << "Optimal cost is " << optCost << std::endl;
    return 0;
}
```

Calculating optimal cost

```
MTP.cpp
// Note : must be declared before main() function
// hw, vw : horizontal and vertical input weights
// cost : stored optimal cost from (0,0) to (r,c)
// move : stored optimal decision to reach (r,c)
// r,c : the position of interest
int optimalCost(Matrix615<int>& hw, Matrix615<int>& vw, Matrix615<int>& cost,
    Matrix615<int>& move, int r, int c) {
    if ( cost.data[r][c] == 0 ) { // if cost is stored already, skip
        if ( ( r == 0 ) && ( c == 0 ) ) cost.data[r][c] = 0; // terminal condition
        else if ( r == 0 ) { // only horizontal move is possible
            move.data[r][c] = 0; // 0 means horizontal move to (r,c)
            cost.data[r][c] = optimalCost(hw,vw,cost,move,r,c-1) + hw.data[r][c-1];
        }
        else if ( c == 0 ) { // only vertical move is possible
            move.data[r][c] = 1; // 1 means vertical move to (r,c)
            cost.data[r][c] = optimalCost(hw,vw,cost,move,r-1,c) + vw.data[r-1][c];
        }
    }
}
```


Calculating optimal cost (cont'd)

```
MTP.cpp
else { // evaluate the cumulative cost of horizontal and vertical move
    int hcost = optimalCost(hw,vw,cost,move,r,c-1) + hw.data[r][c-1];
    int vcost = optimalCost(hw,vw,cost,move,r-1,c) + vw.data[r-1][c];
    if ( hcost > vcost ) { // when vertical move is optimal
        move.data[r][c] = 1; // store the decision
        cost.data[r][c] = vcost; // and store the optimal cost
    }
    else {
        move.data[r][c] = 0;
        cost.data[r][c] = hcost;
    }
}

// when horizontal move is optimal
return cost.data[r][c]; // return the optimal cost }
}
```

Dynamic programming : A smart recursion

- Dynamic programming is recursion without repetition
 - Formulate the problem recursively
 - Build solutions to your recurrence from the bottom up
- Dynamic programming is not about filling in tables; it's about smart recursion (Jeff Erickson)

Minimum edit distance problem

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

An example
FOOD → MOOD → MON_^D → MONED → MONEY
 Edit distance is 4 in the example above

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

- Similar representation to DNA sequence alignment
- Does the above alignment provides an optimal edit distance?

Summary

Today

- Dynamic programming is a smart recursion avoiding redundancy
- Divide a problem into subproblems that can be shared
- Examples of dynamic programming
 - Fibonacci numbers
 - Manhattan tourist problem
 - Overview of edit distance problem

Next lecture

- Edit Distance
- Introduction to Hidden Markov model