Recap: Hash Tables

Key features

- $\Theta(1)$ complexity for Insert, Search, and Remove
- Requires large memory space than the actual content for maintaining good performance
- But uses much smaller memory than direct-address tables

Key components

- Hash function
  - $h(x, key)$ mapping key onto smaller ‘addressible’ space $H$
  - Total required memory is the possible number of hash values
  - Good hash function minimize the possibility of key collisions
- Collision-resolution strategy, when $h(k_1) = h(k_2)$.

Recap: Illustration of ChainedHash

Recap: Open hash

Probing strategies

- Linear probing
- Quadratic probing
- Double hashing

Double Hashing

- $h(k, i) = (h_1(k) + ih_2(k)) \mod m$
- The probe sequence depends in two ways upon $k$.
- For example, $h_1(k) = k \mod m, h_2(k) = 1 + (k \mod m')$
- Avoid clustering problem
- Performance close to ideal scheme of uniform hashing.
**Today**

**Dynamic Programming**
- Fibonacci numbers
- Manhattan tourist problems
- Edit distance problem

**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**

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

**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.

**Performance of recursive FIBONACCI**

**Computational time**
- 4.4 seconds for calculating \( F_{40} \)
- 49 seconds for calculating \( F_{45} \)
- \( \infty \) seconds for calculating \( F_{100} \)!
What is happening in the recursive FIBONACCI

![Fibonacci Diagram]

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

```c
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

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

Dynamic programming

Key components of dynamic programing
- 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 brute-force algorithm

And here comes an optimal solution

A slightly better, but still not an optimal solution

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

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

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

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

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.
Implementing Manhattan tourist algorithm

```cpp
template<class T>
class Matrix { // Matrix data type to store the costs
    T* data; // internal data as one-dimensional array
    int nr, nc; // # rows and # cols
    Matrix(const Matrix<T>& m) {};// prevent copy
    public:
        Matrix(int nrows, int ncols): nr(nrows), nc(ncols) {
            data = new T[nrows*ncols]; // initialize matrix
        }
    Matrix() { if (data != NULL) delete [] data; }
    // accessor function : possible to use to read/write elements
    // value1 = M.at(i,j);
    // M.at(i,j) = value2;
    T& at(int r, int c) { return data[r*nc+c]; }
    void print(); // print the content of the matrix (omitted)
};
```

Calculating optimal cost

```cpp```
int main(int argc, char** argv) {
    int nrows = 5, ncols = 5;
    Matrix<int> hw(nrows,ncols-1), vw(nrows-1,ncols); // weight matrices
    hw.at(0,0) = 4; hw.at(0,1) = 2; // initialize horizontal weights
    vw.at(0,0) = 0; vw.at(0,1) = 6; // initialize vertical weights
    // optimal costs and decisions for backtracking
    Matrix<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;
    // backtrack the stored decision to reconstruct an optimal path
    trackOptimalPath(hw,vw,cost,move,nrows-1,ncols-1);
    return 0;
}
```
Dynamic programming: A smart recursion

- Dynamic programming is recursion without repetition
  1. Formulate the problem recursively
  2. 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  →  MOND  →  MONED  →  MONEY
```

Edit distance is 4 in the example above

More examples of edit distance

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

- Input strings are $x[1,\cdots,m]$ and $y[1,\cdots,n]$.
- Let $x_i = x[1,\cdots,i]$ and $y_j = y[1,\cdots,j]$ be substrings of $x$ and $y$.
- Edit distance $d(x, y)$ can be recursively defined as follows
  \[
  d(x_i, y_j) = \begin{cases} 
  i & j = 0 \\
  j & i = 0 \\
  \min \left\{ 
  d(x_{i-1}, y_j) + 1, \\
  d(x_i, y_{j-1}) + 1, \\
  d(x_{i-1}, y_{j-1}) + I(x[i] \neq y[j]) 
  \right\} & \text{otherwise}
  \end{cases}
  \]
- Similar to the Manhattan tourist problem, but with 3-way choice.
- Time complexity is $\Theta(mn)$.

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
  - Edit distance problem

Next lecture
- Algorithms in graphs
  - Using boost library
  - Dijkstra’s algorithm (CLRS Chapter 24)
- Introduction to hidden Markov model