IS 2610: Data Structures

Recursion, Divide and conquer
Dynamic programming,

Feb 2, 2004
Recursion and Trees

- Recursive algorithm is one that solves a problem by solving one or more smaller instances of the same problem
  - Functions that call themselves
  - Can only solve a base case Recursive function calls itself

- If not base case
  - Break problem into smaller problem(s)
  - Launch new copy of function to work on the smaller problem (recursive call/recursive step)
    - Slowly converges towards base case
    - Function makes call to itself inside the return statement
  - Eventually base case gets solved
  - Answer works way back up, solves entire problem
Algorithm for pre-fix expression

```c
char *a; int i;
int eval()
{
    int x = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
        {i++; return eval() + eval(); }
    if (a[i] == '*')
        {i++; return eval() * eval(); }
    while ((a[i] >= '0') && (a[i] <= '9'))
        x = 10*x + (a[i++]-'0');
    return x;
}
```

Example:
```
eval() * + 7 6 12
  eval() + 7 6
  eval() 7
  eval() 6
  return 13 = 7 + 6
  eval() 12
  return 12 * 13
```
Recursive vs. iterative solution

- In principle, a loop can be replaced by an equivalent recursive program
  - Recursive program usually is more natural way to express computation
- Disadvantage
  - Nested function calls –
    - Use built in pushdown stack
    - Depth will depend on input
    - Hence programming environment has to maintain a stack that is proportional to the push down stack
    - Space complexity could be high
Divide and Conquer

- Many recursive programs use recursive calls on two subsets of inputs (two halves usually)
  - Divide the problem and solve them – divide and conquer paradigm
  - Property 5.1: a recursive function that divides a problem size $N$ into two independent (nonempty) parts that it solves recursively calls itself less than $N$ times
  - Complexity: $T_N = T_k + T_{N-k} + 1$
Find max - Divide and Conquer

```c
int max(int a[], int l, int r)
{
    int m = (l+r)/2;
    if (l == r) return a[l];
    int u = max(a, l, m);
    int v = max(a, m+1, r);
    if (u > v) return u;
    else return v;
}
```

Dynamic programming

- When the sub-problems are not independent, the situation may be complicated
  - Time complexity can be very high
- Example
  - Fibonacci number
    - Base case: $F_0 = F_1 = 1$
    - $F_n = F_{n-1} + F_{n-2}$

```c
int fibonacci(int n){
    if (n<=1) return 1; // Base case
    return fibonacci(n-1) + fibonacci(n-2);
}
```
Recursion: Fibonacci Series

- **Order of operations**
  - `return`
    - `fibonacci(n - 1) + fibonacci(n - 2);`

- **Recursive function calls**
  - Each level of recursion doubles the number of function calls
    - 30th number = $2^{30} \sim 4$ billion function calls
  - Exponential complexity
Simpler Solution

- **Linear!!**

- **Observation**
  - We can evaluate any function by computing all the function values in order starting at the smallest, using previously computed values at each step to compute the current value
  - **Bottom-up Dynamic programming**
    - Applies to any recursive computation, provided that we can afford to save all the previously computed values
  - **Top-down**
    - Modify the recursive function to save the computed values and to allow checking these saved values
      - **Memoization**
Dynamic Programming

- Top-down: save known values
- Bottom-up: pre-compute values
  - Determining the order may be a challenge
- Top-down preferable
  - It is a mechanical transformation of a natural problem
  - The order of computing the sub-problems takes care of itself
  - We may not need to compute answers to all the sub-problems

```c
int F(int i)
{
    int t;
    if (knownF[i] != unknown)
        return knownF[i];
    if (i == 0) t = 0;
    if (i == 1) t = 1;
    if (i > 1) t = F(i-1) + F(i-2);
    return knownF[i] = t;
}
```
Dynamic programming
Knapsack problem

- **Property**: DP reduces the running times of a recursive function to be at most the time required to evaluate the function for all arguments less than or equal to the given argument.

- Knapsack problem
  - **Given**
    - N types of items of varying size and value
    - One knapsack (belongs to a thief!)
  - **Find**: the combination of items that maximize the total value
Knapsack problem

Knapsack size: 17

<table>
<thead>
<tr>
<th>Item</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Val</td>
<td>4</td>
<td>5</td>
<td>10</td>
<td>11</td>
<td>13</td>
</tr>
</tbody>
</table>

int knap(int cap)
{
    int i, space, max, t;
    for (i = 0, max = 0; i < N; i++)
        if ((space = cap - items[i].size) >= 0)
            if ((t = knap(space) + items[i].val) > max)
                max = t;
    return max;
}

int knap(int M)
{
    int i, space, max, maxi, t;
    if (maxKnown[M] != unknown) return maxKnown[M];
    for (i = 0, max = 0; i < N; i++)
        if ((space = M-items[i].size) >= 0)
            if ((t = knap(space) + items[i].val) > max) { max = t; maxi = i; }
    maxKnown[M] = max; itemKnown[M] = items[maxi];
    return max;
}
Trees

- Trees are central to design and analysis of algorithms
  - Trees can be used to describe dynamic properties
  - We build and use explicit data structures that are concrete realization of trees

General issues:
- Trees
- Rooted tree
- Ordered trees
- M-ary trees and binary trees
### Tree

- **Trees**
  - Non-empty collection of vertices and edges
  - Vertex is a simple object (a.k.a. node)
  - Edge is a connection between two nodes
  - Path is a distinct vertices in which successive vertices are connected by edges
    - There is precisely one path between any two vertices
- **Rooted tree:** one node is designated as the root
- **Forest**
  - Disjoint set of trees

![Diagram of a binary tree with labeled nodes and edges](image_url)
Definitions

- Binary tree is either an external node or an internal node connected to a pair of binary trees, which are called the left sub-tree and the right sub-tree of that node
  - Struct node {Item item; link left, link right;}
- M-ary tree is either an external node or an internal node connected to an ordered sequence of M-trees that are also M-ary trees
- A tree (or ordered tree) is a node (called the root) connected to a set of disjoint trees. Such a sequence is called a forest.
  - Arbitrary number of children
    - One for linked list connecting to its sibling
    - Other for connecting it to the sibling
Example general tree
A binary tree with $N$ internal nodes has $N + 1$ external nodes

- Proof by induction
- $N = 0$ (no internal nodes) has one external node
- Hypothesis: holds for $N-1$
- $k, N - 1 - k$ internal nodes in left and right sub-trees (for $k$ between 0 and $N-1$)
- $(k+1) + (N - 1 - k) = N + 1$
Binary tree

- A binary tree with N internal nodes has 2N links
  - $N-1$ to internal nodes
    - Each internal node except root has a unique parent
    - Every edge connects to its parent
  - $N+1$ to external nodes

- Level, height, path
  - Level of a node is 1 + level of parent (Root is at level 0)
  - Height is the maximum of the levels of the tree’s nodes
  - Path length is the sum of the levels of all the tree’s nodes
  - Internal path length is the sum of the levels of all the internal nodes
Examples

- Level of D?
- Height of tree?
- Internal length?
- External length?

- Height of tree?
- Internal length?
- External length?
Binary Tree

- External path length of any binary tree with $N$ internodes is $2N$ greater than the internal path length.
- The height of a binary tree with $N$ internal nodes is at least $\log N$ and at most $N-1$.
  - Worst case is a degenerate tree: $N-1$.
  - Best case: balanced tree with $2^i$ nodes at level $i$.
    - Hence for height: $2^{h-1} < N+1 = 2^{h} -$ hence $h$ is the height.
Binary Tree

Internal path length of a binary tree with $N$ internal nodes is at least $N \ l g \ (N/4)$ and at most $N(N-1)/2$

- Worst case: $N(N-1)/2$
- Best case: $(N+1)$ external nodes at height no more than $\left\lfloor \ l g \ N \right\rfloor$
  - $(N+1) \left\lfloor \ l g \ N \right\rfloor - 2N < N \ l g \ (N/4)$
Tree traversal (binary tree)

- **Preorder**
  - Visit a node,
  - Visit left subtree,
  - Visit right subtree

- **Inorder**
  - Visit left subtree,
  - Visit a node,
  - Visit right subtree

- **Postorder**
  - Visit left subtree,
  - Visit right subtree
  - Visit a node
Recursive/Nonrecursive Preorder

```c
void traverse(link h, void (*visit)(link))
{
    if (h == NULL) return;
    (*visit)(h);
    traverse(h->l, visit);
    traverse(h->r, visit);
}

void traverse(link h, void (*visit)(link))
{
    STACKinit(max);
    STACKpush(h);
    while (!STACKempty())
    {
        (*visit)(h = STACKpop());
        if (h->r != NULL) STACKpush(h->r);
        if (h->l != NULL) STACKpush(h->l);
    }
}
```
Recursive binary tree algorithms

- Exercise on recursive algorithms:
  - Counting nodes
  - Finding height
Sorting Algorithms

- Selection sort
  - Find smallest element and put in the first place
  - Find next smallest and put in second place
  - ..
  - Try out! Complexity? Recursive?

- Bubble sort
  - Move through the elements exchanging adjacent pairs if the first one is larger than the second
  - Try out! Complexity?
Insertion sort

```
// People

#define less(A, B) (key(A) < key(B))
#define exch(A, B) { Item t = A; A = B; B = t; }
#define compexch(A, B) if (less(B, A)) exch(A, B)

void insertion(Item a[], int l, int r) {
    int i;
    for (i = l+1; i <= r; i++)
        compexch(a[l], a[i]);
    for (i = l+2; i <= r; i++) {
        int j = i; Item v = a[i];
        while (less(v, a[j-1])) {
            a[j] = a[j-1]; j--;
        }
        a[j] = v;
    }
}

#define define less(A, B) (key(A) < key(B))
#define define exch(A, B) { Item t = A; A = B; B = t; }
#define define compexch(A, B) if (less(B, A)) exch(A, B)
```