Algorithms!

Meichen Lu (ml574@cam.ac.uk)

May 1, 2018

Contents

1 Overview

What is an algorithm?

- Mathematical abstraction of computer program
- Computational procedure to solve a problem

Analogues between computing and algorithm

- Program VS algorithm
- Programming language VS pseudocode (structured English)
- Computer VS model of computation

1.1 Model of computation

Model of computation specifies

- what operations an algorithm is allowed
- cost (time, space, . . .) of each operation
- cost of algorithm = sum of operation costs

Random Access Machine (RAM)

Pointer machine

Python machine

1.2 Example of acceleration

Document Distance Algorithm

1. split each document into words
2. count word frequencies (document vectors)
3. compute dot product & divide

Different versions!

1. Basic (with insertion sort)
2. Using `word_list.extend(words_in_line)` instead of `words += words on line`
3. Use sort in step (3)
4. use dictionary to count the word frequencies
5. Use `translation_table = string.maketrans(string.punctuation+string.uppercase, " "*len(string.punctuation)+string.lowercase)` to map upper case to lower case and punctuation to spaces
6. merge sort instead of insertion
7. Use dictionary in step (3)
8. Process the whole document instead of line-by-line

2 Foundation

2.1 Divide and conquer

2.1.1 Recursion tree

The recursion tree is demonstrated in Fig 1. The height of the tree is $log_4 n$ (with $log_4 n + 1$ levels). At the bottom level, depth $log_4 n$, there are $3^{log_4 n} = n^{log_4 3}$ nodes.
Total cost over all levels are

$$\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + ... + \left(\frac{3}{16}\right)^{log_4 n-1} cn^2 + \Theta(n^{log_4 3}) \\
&= \sum_{i=0}^{log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{log_4 3}) \\
&= \frac{16}{13}cn^2 + \Theta(n^{log_4 3}) = O(n^2)
\end{aligned}$$

3 Sort

- Heap sort: 4.1.2

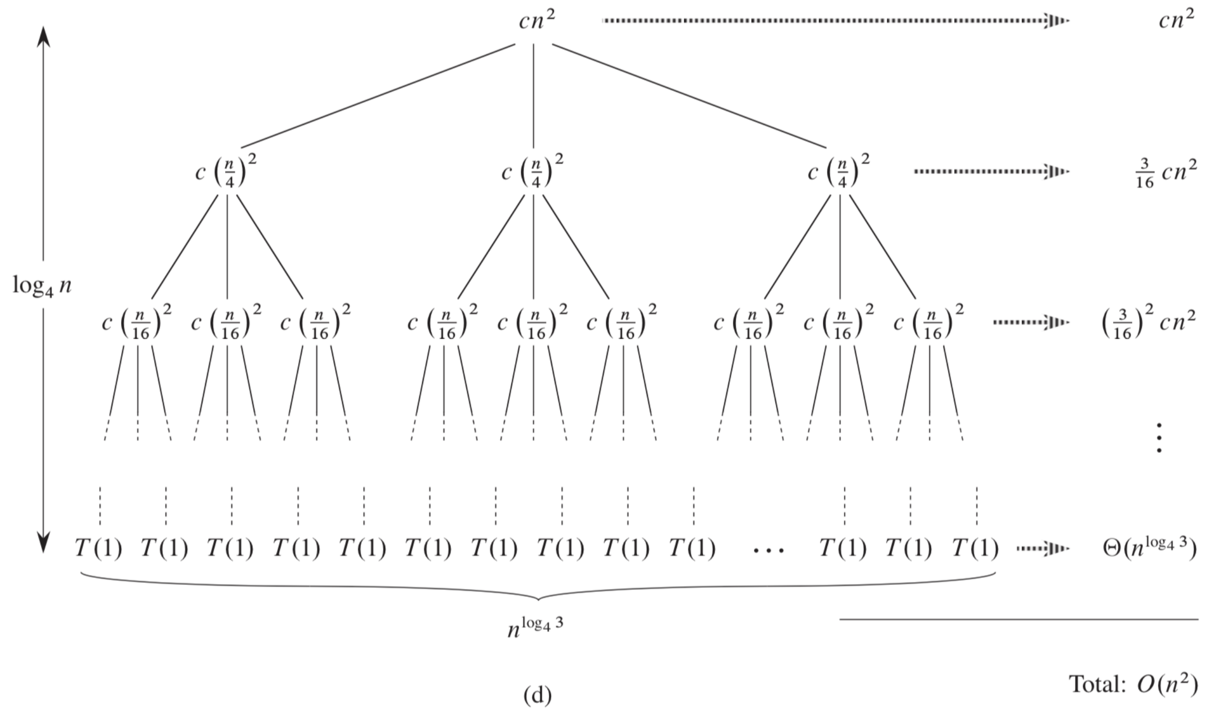

Figure 1: Recursion tree for the recurrences $T(n) = 3T(n/4) + cn^2$

4 Data structures

4.1 Heap

4.1.1 Definition

Heap as a tree

- Root of heap $i = 1$
- For node i in the heap, its parent is element $i/2$
- Its children are left$(i) = 2i$ and right$(i) = 2i + 1$

Max-heap property: the key of a node is $\geq$ the key of its children.

4.1.2 Maxi-heap operations

maxHeapify

- Assume that the trees rooted at left (i) and right (i) are max-heaps
- If element $A[i]$ violates the max-heap property, correct violation by “trickling” element $A[i]$ down the tree, making the subtree rooted at index i a max-heap

- Complexity $O(\lg n)$

buildMaxHeap based on maxHeapify

```
for i = n/2 down to 1:
    maxHeapify(A,i)
```

- The rationale is that $A(i)$ for $i > n/2$ are all leaf nodes
- Complexity $O(n)$

heapSort

1. buildMaxHeap(unordered array)
2. find the maximum element $A[1]$
3. Swap $A[1]$ with the n^{th} (last) element, and delete the last element form the heap.
4. maxHeapify(A,1)

Each step has complexity $O(\lg n)$ so the overall complexity is $O(n \lg n)$

4.2 Binary Search Tree

4.2.1 Basic BST

- Components: each node has a key, left pointer, right pointer and root pointer
- Properties: all the elements to the left of a node is smaller than the node value, and all elements to the right are larger.
- BSTs support insert, delete, min, max, next-larger, next-smaller, etc. in $O(h)$ time, where h = height of tree (= height of root)
- Height of the tree is the longest path going down from the root to the leaf: $\lg n \leq h \leq n$

BST augmentation Each node stores the subtree size (the number of node within at the node and its branches)

4.3 Balanced BST

balanced BST maintains $h = O(\lg n)$ such that all operations run in $O(\lg n)$ time.

AVL trees requires the height of the left & right child of the node to be ± 1

Rotate

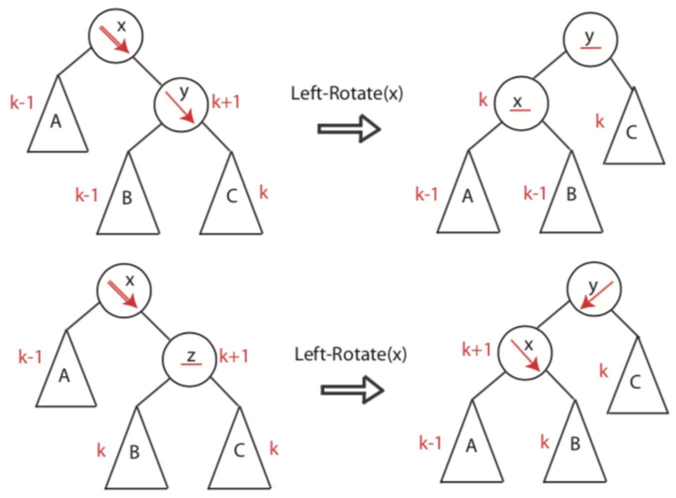

Figure 2: AVL balancing

AVL insert

1. Simple BST insert
2. Fix AVL property from the changed node up

4.4 Dictionary and hashing

Basic operations for dictionary

- Insert with overwrite (`dict[key] = val`)
- Search for key (`dict[key]`)
- Delete key

To execute the basic operations efficiently, we use a **Direct Access Table**. This means items would need to be stored in an array, indexed by key (random access)

4.4.1 prehash

The key are not always non-negative integers, so we use prehash (in python the function is called `hash`)

- In theory, $x = y \Leftrightarrow \text{prehash}(x) = \text{prehash}(y)$
- The keys should not be mutable (e.g. lists)

4.4.2 hash

Reduce universe $\mathcal{U}$ of all keys (say, integers) down to reasonable size m for table. ('hash means chop things up, mix them and do all kinds of weird things, as in hash potato')

$$\text{hash function h: } \mathcal{U} \rightarrow \{0, 1, 2, ..., m-1\}$$

Collision What to do when two keys map to the same integer?

Chaining Store the key as a linked list

Open addressing Another approach to collisions

- One item per slot
- Hash function specifies the order of slots to probe (try) for a key (for insert/search/delete) $\langle h(k,0), h(k,1), ..., h(k,m-1)\rangle$
- Tricks with delete operation: replace item with 'DeleteMe' flag
- Probing strategy: linear probing can cause 'clustering' problem; double hashing $h(k,i) = (h_1(k) + ih_2(k)) \mod m$

Simple uniform hashing An assumption: Each key is equally likely to be hashed to any slot of table, independent of where other keys are hashed. Expected chain length for n keys to be stored in m slots in the table is $\alpha = \frac{n}{m}$ Thus the expected computing cost is $1 + \alpha$

5 Numerics

5.1 Root-finding and Newton's method

5.2 High-precision multiplication

Multiplying two n-digit numbers with radix r Rewrite the digits into two halves

$$x = x_1 \cdot r^{n/2} + x_0$$
$$y = y_1 \cdot r^{n/2} + y_0$$

Naive way of doing multiplication

$$z = x \cdot y = x_1 y_1 \cdot r^n + (x_0 y_1 + x_1 y_0) \cdot r^{n/2} + x_0 y_0$$

Complexity is $\Theta(n^2)$

Karatsuba's method Doing three multiplies instead of 4

$$\begin{aligned}
z_0 &= x_0 y_0 \\
z_2 &= x_1 y_1 \\
z_1 &= (x_0 + x_1)(y_0 + y_1) - z_0 - z_2 \\
&= x_0 y_1 + x_1 y_0 \\
z &= z_2 \cdot r^n + z_1 \cdot r^{n/2} + z_0
\end{aligned}$$

Complexity $\Theta(d^{\log_2 3}) = \Theta(d^{1.584\ldots})$
Other methods exist and approaches $\Theta(d)$, we say the complexity of multiplication is $\Theta(d^\alpha)$

5.3 High precision division

For $\frac{R}{b}$ where R is a large value s.t. it is easy to divde by R, e.g. $R = 2^k$. Use Newton's method for computing $\frac{R}{b}$.

$$\begin{aligned}
f(x) &= \frac{1}{x} - \frac{b}{R} \\
f'(x) &= -\frac{1}{x^2} \\
x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} = 2x_i - \frac{bx_i^2}{R}
\end{aligned}$$

By error analysis $\epsilon_{i+1} = -\epsilon_i^2$, we have quadratic convergence and the number of accurate digits double at each step. Complexity analysis: for $n - digit$ precision, we only need n^α operations for the multiplication. We have the number of operations to be

$$c \cdot 1^\alpha + c \cdot 2^\alpha + c \cdot 4^\alpha + \cdots + c \cdot (\frac{d}{2})^\alpha + c \cdot d^\alpha < 2c \cdot d^\alpha$$

6 Graphs

6.1 Graph representations

Graph $G = (V, E)$ where V = set of verticies and E = set of edges, including ordered pair (a, b) for directed graph or unordered pair a, b for undirected graph.

- Adjacency lists
  - For each vertex $u \in V$, $Adj[u]$ stores u's neighbors for undirected/outgoing edges for directed graphs.
  - In Python, vertex is any hashable object and Adj is dictionary of list/set values
- Implicit graphs: compute on the fly
- Object-oriented variations (object can be each vortex or each edge)

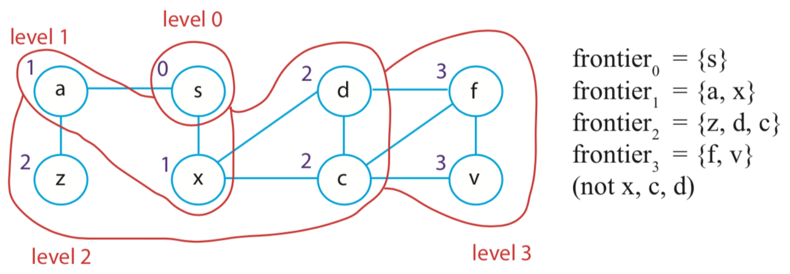

Figure 3: Breadth-first search (BFS) for a graph

6.2 Applications

- Web crawling
- Social networking
- network broadcast routing
- garbage collection
- model checking (finite state machine)
- checking mathematical conjectures
- solving puzzles and games

6.3 Breadth-first search

- Results stored in the dictionary e.g. level = $\{\ s : 0\ \}$
- The algorithm runs through a loop visiting the $Adj[frontier]$ and check if the the neighbour is in the dictionary level. If not, the element is added to the dictionary.

$$\text{time} = \sum_{v\in V} |Adj[V]| = \begin{cases} |E|, \text{for directed graphs,} \\ 2|E|, \text{for undirected graphs.} \end{cases}$$

- Run time $O(V + E)$.
- Shortest path to go from the root has length = level$[v]$
- Use parent pointers to find one of the shortest-path.

6.4 Depth-first search

- Recursively explore and follow path until you get stuck
- Run time for the recursive algorithm starting from a vertex s is time = $\sum_{v\in V} |Adj[V]| = O(E)$ as before and outer loop add $O(V)$.
- Run time $O(V + E)$.

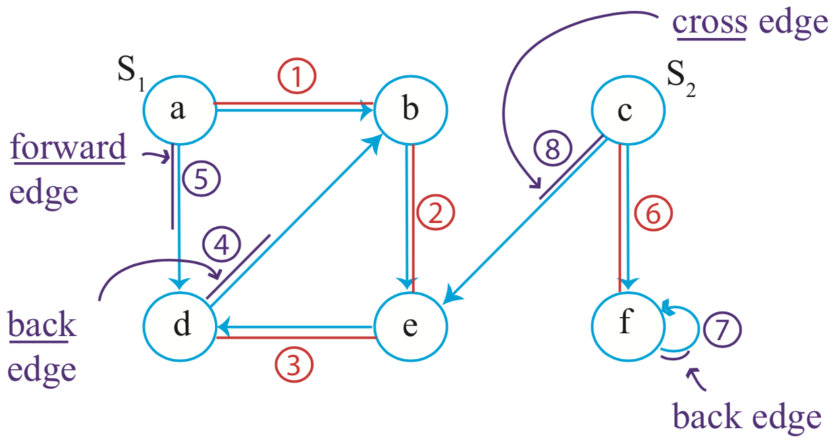

Figure 4: Edge classification based on the depth-first search (DFS) of a graph

- Benefit: edge classification: see Fig. 4

Edge classification

- Tree edge: edges formed during the search
- Forward edge: to descendant
- Back edge: to ancestor
- Cross edge: to another subtree

In a undirected graph, there are only tree edges and backward edges.

Cycle detection Graph G has a cycle $\Leftrightarrow$ DFS has a back edge.

Job scheduling for a given Directed Acylic Graph (DAG), where vertices represent tasks & edges represent **dependencies**, order tasks without violating dependencies. We use **topological sort** by reverse of DFS finishing times (time at which DFS-Visit(v) finishes).
Proof of correctness: for any edge (u, v),u is ordered before v, i.e., v is finished in DFS before u (by finishing, it means fully exploring the subtree rooted at u).

- If u visited before v, will visit v as the edge or otherwise. Thus v finished before u.
- If v visited before u, u cannot finish before v otherwise there would be a loop.

6.5 Shorted paths with weighted graphs

Motivations

- Shortest path on a map
- Cost to complete a task

- Closest friend ‘path’ on facebook

Notations

- shortest path from s to v is $\delta(s,v)$. If there is no such path $\delta(s,v)=\infty$
- $d[v]$ is one of the path lengths to v, $d[s]=0$
- $\Pi[v]$ is predecessor on best path to v, $\Pi[s]=NIL$

6.5.1 Shortest path algorithm (no negative cycles)

Initialising all weights to ∞ and all edges from a vertex at a time.
Basic operation is the relaxation operation. Proof that **relaxation is safe**
By definition, $d[u]\geq\delta(s,u)$ and by triangle inequality, $\delta(s,v)\leq\delta(s,u)+\delta(u,v)$. Thus $\delta(s,v)\leq$ $[u]+w(u,v)$. So letting $d[v]=d[u]+w(u,v)$ is safe i.e., $d[v]\geq\delta(s,v)$ holds for all $v\in V$

Implementation with DAG

1. Topologically sort the DAG
2. One pass over vertices in topologically sorted order. $\Theta(V+E)$ time.
3. N.B. all vertices to the left of s will be ∞ as they are not reachable from s

6.5.2 Dijkstra’s algorithm

For each edge $(u,v)\in E$, assume$w(u,v)\geq 0$, maintain a set S of vertices whose final shortest path weights have been determined. Repeatedly select $u\in(VS)$ with minimum shortest path estimate, add u to S, relax all edges out of u.
In short, **relaxation from the shortest path identified so far**

- the priority queue Q takes all the element
- S is a list with all the vertices already sorted.
- Complexity: $\Theta(v)$ inserts into priority queue, Q, $\Theta(v)$ extract_min operation on Q, $\Theta(E)$ decrease_key operation from Q

Acceleration: bi-directional search Bi-directional search means alternative forward search from s and backward search from t until some vortex w has been removed from the queues of both searches Q_f and Q_b.
Subtlety After search terminates, find node x with minimum value of $d_f(x)+d_b(x)$ for any x that has been deleted from either Q_f or Q_b or both.
N.B. It does not change worst-case behavior, but reduce the number of visited vertices in practice.

Acceleration: goal-directed search with potential function Modify edge weights with potential function over vertices but the shortest paths are maintained. The intuition is that with modified weights, the search can be accelerated.
One example is to use landmarks as in a map.

6.5.3 Bellman-Ford algorithm

Pseudo code:

```
Initialise()
# Find the shortest path
for i = 1 to |V|-1:
    for each edge (u,v) in E: # update for all edges
        Relax(u,v)
# Check for negative-weighted cycle
for each edge (u,v) in E:
    do if d[v]>d[u] + w(u,v):
        then report a negative-weighted cycle exists
```

Proof of Bellman-Ford algorithm

1. Consider path $p = \langle v_0, v_1, ..., v_k \rangle$ from v_0 to $v_k = v$ that is a shortest path with minimum number of edges. p must not have negative weight cycle. $\Rightarrow p$ is simple $\Rightarrow k \leq |V| - 1$.
2. After 1 pass through E, we have $d[v_1] = \delta(s, v_1)$ because we will relax the edge (v_0, v_1) and cannot find a shorter path than this (optimal substructure and safeness lemma)
3. After i passes through E, we have $d[v_1] = \delta(s, v_i)$
4. After i passes through E, we have $d[v_k] = d[v] = \delta(s, v)$

7 Dynamic Programming

- Powerful algorithmic design technique
- Smart brute force
- DP = recursion + memoization (remember) + guessing
- DP = dividing into reasonable # subproblems whose solutions relate *acyclicly* usually via guessing parts of solution.

7.1 Fibonnaci numbers and two different perspectives

Naive recursion

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

$T(n) = T(n-1) + T(n-2) + O(1) \geq 2T(n-2) + O(1) \geq 2^{n/2}$ Exponential - bad!

Memoized DP algorithm

```
memo = {}
fib(n):
    if n in memo: return memo[n]
    else: if n <= 2: f = 1
        else: f = fib(n-1) + fib(n-2)
        memo[n] = f
        return f
```

- The key is to remember and reuse!
- $fib(k)$ only recurses first time called for any k while is read from the memory for all other calls (memoized call).
- memoized calls are free: $\Theta(1)$ time
- total time = # of subproblems $\cdot$ time per subproblem $= n\Theta(1) = \Theta(n)$

Bottom-up DP algorithm

```
fib = {}
for k in [1, 2, ... , n]:
    if n <= 2: f = 1
    else:
        f = fib(k-1) + fib(k-2)
        fib[k] = f
return fib[n]
```

- Exactly the same computation as memoized DP
- Solving the subproblems in the right order (topological sort of subproblem dependency DAG)

7.1.1 Shortest paths with DP

Guessing To find the shortest path to v try all choices and use the best one u and solve the subproblem $\delta(s, u)$

- Recursive formulation $\delta(s, v) = min\{w(u, v) + \delta(s, u) | (u, v) \in E\}$
- Memoized DP algorithm does not work because it takes **infinite time** if there are cycles!
- Works for directed graphs in $O(V + E)$: DFS/topological sort + Bellman-Ford round rolled into a single recursion
- Can convert the diagram with loops to a DAG by introducing another dimension in time. Then find the shortest path with length $\leq |V|$

7.1.2 Five steps to DP

1. Define subproblems
2. Guess (part of solution)
3. Relate subproblem solutions
4. Recurse + memoize **OR** build DP table bottom-up, check subproblems acyclic/topological order
5. Solve original problem **OR** combine subproblem solutions

Parent pointer To identify the optimal path, use parent pointer.

Text justification similar problems include blackjack with perfect information

1. **subproblem** = minimum badness for suffix words[i :] $\Rightarrow$ # subproblems = $\Theta(n)$ where n= # words
2. **guessing** = where to end first line, say $i : j \Rightarrow$ # choices = $n = 1 = O(n)$
3. **recursion** $DP[i] = \min(\text{badness}(i, j) + DP[j]$ for j in range $(i+1, n+1))$ and $DP[0] = 0 \Rightarrow$ time per subproblem $\Theta(n)$
4. **topological order**: for $i = n, n-1, ..., 1, 0$ total time $\Theta(n^2)$
5. **original problem**: $DP[0]$ (use parent pointer to find where to cut the text).

7.1.3 Strategy to divide subproblems

- Prefixes: $x[: i] \Rightarrow \Theta(|x|)$
- Suffixes: $x[i :] \Rightarrow \Theta(|x|)$
- Substrings: $x[i : j] \Rightarrow \Theta(x^2)$

Examples include: parenthesization, text editing, knapsacking.

Parenthesization refers to the optimal way to evaluate matrices

1. **subproblem** = cost of substring $A[i : j] \Rightarrow$ # subproblems = $\Theta(n^2)$ where n= # matrices
2. **guessing** = outermost multiplication $\Rightarrow$ # choices = $O(n)$
3. **recursion** $DP[i, j] = \min(DP[i, k]+DP[k, j]+$ cost for multiplying $(A[i] \cdots A[k-1])$ by $(A[k] \cdots A[j-1])$ for k in range $(i+1, j))$ and $DP[i, i+1] = 0 \Rightarrow$ time per subproblem $\Theta(n)$
4. **topological order**: for increasing substring size. Total time $O(n^3)$
5. **original problem**: $DP[0, n]$ (use parent pointer to find where to put parenthesis).

7.1.4 Two kinds of guessing

- First kind of guessing is which subproblem to use (except Fibonacci)
- Second kind of guessing is to create more subproblems

Examples include piano fingering, simplified Tetris, Super Mario Bros

Piano fingering Assume we play a single note at each time with the right hand for a musical piece with n notes. Given difficulty to transit between two nodes $d(f, p, g, q)$, where f, g are fingers and p, q are notes before and after the transition.

1. **subproblem** = difficulty for suffix notes[i :] given finger f on first notes[i] $\Rightarrow$ # subproblems $= \Theta(nF)$ where $n=$ # notes with F fingers
2. **guessing** = finger g for the next note[$i+1$] $\Rightarrow$ # choices $= O(F)$
3. **recursion** $DP[i, f] = \min(DP[i+1, g] + d(f,\ \text{note}[i], g,\ \text{note}[i+1])$ for g in range $(F))$ and $DP[n, f] = 0 \Rightarrow$ time per subproblem $\Theta(F)$
4. **topological order**:
```
for $i$ in reverse(range($n$)):
    for $f$ in $1, 2, ...F$:
```
   $\Rightarrow$ total time $O(nF^2)$
5. **original problem**: $\min(DP[0, f]$ for f in $1, 2, ...F$ (use parent pointer to find where to put parenthesis).