Prepared by: [Arun Konagurthu]

Source material acknowledgement

These lecture frames are built on the [source material] developed by [Lloyd Allison] at Monash University.

FIT2004 S1/2017: Algorithms and Data Structures

Week 12, Lecture:

(1) Recursion (2) Algorithm Design Principles

Faculty of Information Technology, Monash University

What is covered in this lecture?

The last week's lectures will cover topics on:

- Recursion
  - Linear Recursion
  - Binary Recursion
  - N-ary Recursion
- Contextualizing Recursion based on various algorithms we dealt in this unit (and some extra ones)
- Various Algorithmic "design principles" encountered in this unit.
- Best of Luck!

Recommended reading

- [Recursion]
- [Linear Recursion]
- [Binary Recursion]
- [N-ary Recursion]

Part 1: Recursion

Linear Recursion

The simplest form of recursion is linear recursion. It occurs where an algorithm action has a simple repetitive structure consisting of some basic step followed by the action again.

Linear Recursion – setup

```
1 function linearRecursiveFunction( x ) {
2     if( base_case is true )
3         process(x); // process base case
4     else {// recursive case
5         something1( x ); // do something with x
6         linearRecursiveFunction( f(x) );
7         something2( x ); // do somthing else with x
8     }
9 }
```

Initial call from some driver routine: **linearRecursiveFunction(y)**

Linear Recursion – execution sequence

```
something1(y)
    something1(f(y))
        something1(f(f(y))
            ...
                ...
                    until base_case is true, where
                    f(f(...f( y ))) is processed
                ...
            ...
         something2(f(f(y))
     something2(f(y))
something2(y)
```

Time Complexity that one often sees with Linear Recursion

```
1    T_0 = a  // base case time
2    T_n = b + T_{n-1}  // Time recurrence relation
3
4    /* Solution of this recurrence can be given
5       using induction -- Refer week 1 lecture slides
6    T_n = a + b*n
```

What assumptions are being made here?

Time Complexity that one often sees with Linear Recursion

```
1    T_0 = a  // base case time
2    T_n = b + T_{n-1}  // Time recurrence relation
3
4    /* Solution of this recurrence can be given
5       using induction -- Refer week 1 lecture slides
6    T_n = a + b*n
```

What assumptions are being made here?

Assumes the evaluation of the **base_case**, **something1()**, **something2()** and **process()** are $O(1)$-time operations

Linear Recursion – Example: Integer exponentiation (x^n)

Exponentiation is the operation of raising one quantity to the power of another.

```
1  //PRECONDITION: x > 0
2  //INPUT: base=x and exponent=n
3  function expn(x, n) {
4      if (n == 0) return 1;
5      if (n < 0 ) return 1/expn(x,-n);
6      if( n is even ) {
7          variable half_n = n/2;
8          variable y=expn(x,half_n);
9          return y*y;
10     }
11     else { // if n is odd
12         return x*expn(x,n-1);
13     }
14 }
```

What is the time-complexity of **expn(x,n)**?

Linear Recursion – Example: Integer exponentiation (x^n)

Exponentiation is the operation of raising one quantity to the power of another.

```
1  //PRECONDITION: x > 0
2  //INPUT: base=x and exponent=n
3  function expn(x, n) {
4      if (n == 0) return 1;
5      if (n < 0 ) return 1/expn(x,-n);
6      if( n is even ) {
7          variable half_n = n/2;
8          variable y=expn(x,half_n);
9          return y*y;
10     }
11     else { // if n is odd
12         return x*expn(x,n-1);
13     }
14 }
```

What is the time-complexity of **expn(x,n)**?

The time-complexity is $O(log(n))$-time. Same as saying it is **linear** in the **length** of (representing) n – i.e., **the number of bits** to store n.

Binary Recursion – setup

A binary-recursive routine (potentially) calls itself twice.

```
1  function binaryRecursiveFunction( x ) {
2      if( base_case is true )
3          process(x); // process the base case
4      else {// recursive case
5          something1(x); // do something with x
6          binaryRecursiveFunction( f(x) );
7          something2(x); // do somthing else with x
8          binaryRecursiveFunction( g(x) );
9          something3(x); // do something other with x
10     }
11 }
```

Initial call from some driver routine: **binaryRecursiveFunction(y)**

Time Complexity of Binary Recursion (under some assumptions)

```
1    T_0 = a  // base case time
2    T_n = b + 2*T_{n-1}  // Time recurrence relation
3
4    /* Solution of this recurrence can be given
5       using induction
6       -- Refer question 4 in week 3 tute sheet */
7       T_n = (a+b)2^n - b
```

Binary Recursion – Example: *n*th Fibonacci number

```
1  function fibonacci(n) {
2      if (n <= 2 ) return 1;
3      else {
4          variable val1 = fib(n-1);
5          variable val2 = fib(n-2);
6          return val1 + val2;
7      }
8
9  }
```

Refer Question 2 in Week 5 Tute sheet!

Binary Recursion Example – Merge Sort (see pre-req Week 3 material on Moodle)

```
1 function merge(int inpA[...], int lo, int hi, int outA[...]){
2 /* sort (input) inpA[lo...hi] into (output) outA[lo...hi] */
3   int i, j, k, mid;
4   if(hi > lo) /* at least 2 elements */
5    { int mid = (lo+hi)/2;             /* lo <= mid < hi */
6
7      merge(outA, lo,   mid, inpA);   /*   sort the ... */
8      merge(outA, mid+1, hi, inpA);   /*    ... 2 halfs */
9
10     /* and now merge them */
11     i = lo;  j = mid+1;  k = lo;
12     while( ... ) {
13        ... merge the sorted inpA[lo...mid] and inpA[mid+1...hi]
14        ... into outA[lo...hi]
15      }/*while */
16   }/*if  */
17 }/*merge */
18
19 function mergeSort(int a[], int N) {  /* wrapper routine */
20 /* NB sorts a[1..N] */
21   int i;
22   int b[N];          /* -- the O(N) workspace */
23
24   for(i=1; i <= N; i++)
25      b[i]=a[i];      /* -- copy */
26
27   merge(b, 1, N, a); /* -- does the real work . . . */
28 }
```

N-ary Recursion

The most general form of recursion is *N*-ary recursion where *N* is **NOT** a constant but **some parameter**. Recursive calls of this kind are useful in generating combinatorial objects (eg. generating all permutations).

N-ary recursion – Example

```
1 function  permute(unused, perm, m, n) {
2    if(m > n) {
3      print permutation in perm
4    }
5    else {
6      for (each elem in unused) {
7        perm[m] = elem;
8        still_unused = unused - {elem}
9        permute(still_unused, perm, m+1, n);//recursion
10     }
11   }
12 }
```

Initial call from some driver routine:

```
permute( {a,b,c...}, [-,-,-,...], 1, n );
```

Permutation – execution sequence

```
permute({a,b,c}, [-,-,-], 1, 3)
  1: permute({b,c}, [a,-,-], 2, 3)
      1.1: permute({c}, [a,b,-], 3,3)
         1.1.1: permute({}, [a,b,c], 4, 3) --> perm=[a,b,c]
      1.2: permute({b}, [a,c,-], 3,3)
         1.2.1: permute({}, [a,c,b], 4, 3) --> perm=[a,c,b]
  2: permute({a,c}, [b,-,-],  2, 3)
      2.1: permute({c}, [b,a,-], 3,3)
         2.1.1: permute({}, [b,a,c], 4, 3) --> perm=[b,a,c]
      2.2: permute({a}, [b,c,-], 3,3)
         2.2.1: permute({}, [b,c,a], 4, 3) --> perm=[b,c,a]
  3: permute({a,b}, [c,-,-], 2, 3)
      3.1: permute({b}, [c,a,-], 3,3)
         3.1.1: permute({}, [c,a,b], 4, 3) --> perm=[c,a,b]
      3.2: permute({a}, [c,b,-], 3,3)
         3.2.1: permute({}, [c,b,a], 4, 3) --> perm=[c,b,a]
```

Note: There are also iterative, non-recursive routines to generate all permutations.

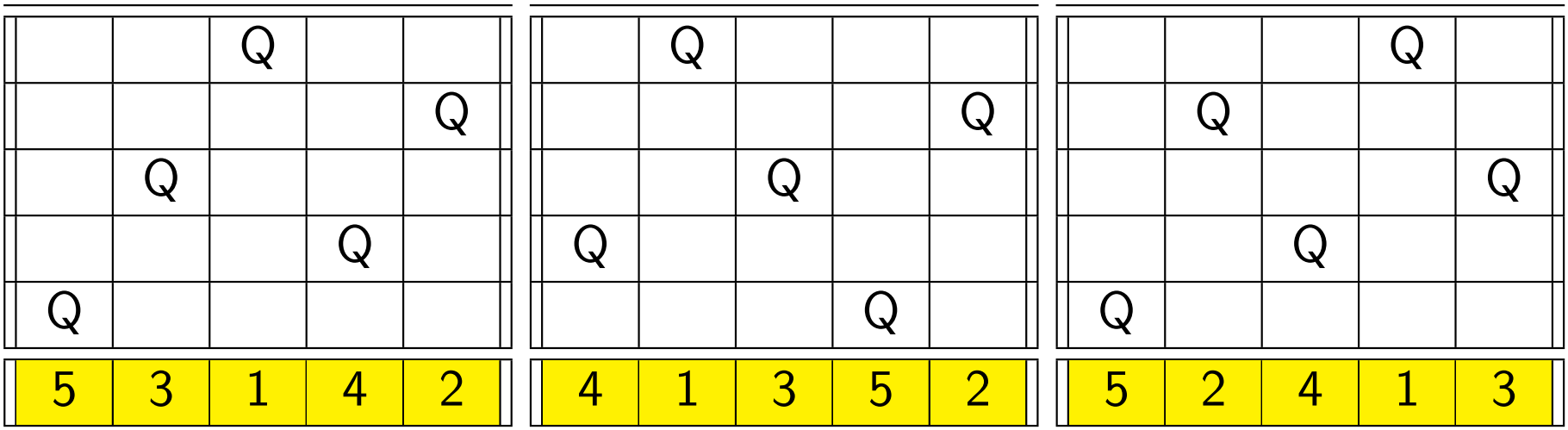
Another N-ary recursion example – N-Queens problem

The N-queens problem is a classic combinatorial problem: place N queens on an $N \times N$ chess board so that no two queens confront each other.

This implies

- No two queens can be on the same row, column or diagonal.
- There must be a queen on each column and all their row numbers must differ.
- Therefore, a solution can be represented as a **`permutation of the rows`**.
- Not all permutations are solutions, NOTE!!!

5-Queens Solutions



... and 7 other possible solutions.

Since we are representing an N-Queens solution as a permutaion, a permutation generator can be turned into an N-queens solver by checking for diagonal conflicts.

N-Queens Pseudocode (as a modification of Permute)

```
1    function NQueens(unused, board, col, N) {
2        if (col > N) print board;
3        else { // if col <= N
4            for (each row in unused) {
5                variable safe = true;
6                for (c in 1..col-1) {      //safety check of
7                    if  ( row = board[c]+col-c //...diagonals
8                       or row = board[c]-col+c ) {
9                        safe = false; break
10                   }
11               }
12               if (safe) {
13                 board[col] = row;
14                 still_unused = unused-{row};
15                 Queens(still_unused, board, col+1, N) //recursion
16               }
17           }
18       }
19   }
```

Initial call to the function takes the form

Queens({1..N}, board, 1, N)

N-Queens Explanation

- The pseudocode in the previous slide modifies in a simple way the permutation generator given on Slide 17.
- At each stage a partial solution, **`board[1..col-1]`**, is safe and an attempt is made to place a queen in column **`col`**.
  - If this is impossible the routine returns.
  - If it is possible and the new queen threatens no previous queen then the extended solution is **`safe`** and a recursive call is made to place the remaining queens.
- The process succeeds when all queens are placed.
- Solutions exist for all $N > 3$, and in fact more than one solution for each value of **`N`**.

This is an example of `back-tracking`

A search for solutions to a combinatorial problem carried out in this way is known as **`back-tracking`**. This comes from the way that partial solutions are extended, backing up at blind alleys where the constraints cannot be satisfied, until one or more complete solutions are found.

Recursion: Summary

- Recursion is a powerful technique
  - giving short algorithms for difficult problems;
  - it is often easy to prove the algorithms correct.
- Linear recursion is the simplest form of recursion
- Binary recursion
  - beware - may take exponential time (but **NOT** necessarily)
  - must introduce a stack to remove recursion.
- *N*-ary recursion useful to generate combinatorial objects

Part 2: Design Techniques (Summing most of the things up!)

Algorithmic Design Techniques

Here, some broad strategies to (**try to**) solve algorithmic problems:

- Look out for good invariants to exploit
- Attempt to balance your work as much as possible
- Do not repeat work (so, store and re-use!)
- Use appropriate data representations
- Try well-known problem solving strategies
- **Sometimes** greed is good!

These are general guidelines. As always, there are many exceptions

Lookout for good invariants to exploit

Here are **some** algorithms we considered in the unit that do precisely this!

- Binary Search (Refer Week 1 lectures)
- Sorting (Refer Lectures from Weeks 2 and 3)
- Shortest Paths and Connectivity
  - Dijkstra's algorithm (Refer Week 8 Lectures)
  - Warshall's and (related) Floyd's algorithm (Refer Week 9 lectures)
- Minimum Spanning Tree Algorithms (Refer Week 10 lectures)

Balance your work as much as possible

For problems that allow division of labour (eg. Divide and Conquer)

- Try to divide work **equally** as much as possible
- Merge sort achieves this (Refer week 3 Prereq material)
  - $O(n \log(n))$-time **always!**
- Quick sort does not necessarily achieve this – depends on the choice of the median (Refer week 3 Lecture 2)
  - Good estimates of the median give $O(n \log(n))$-time
  - Bad estimates of the median give $O(n^2)$-time

Choose data structures with care

Certain data representations are more efficient than others for a given problem

- Priority Queue in Dijkstra's algorithm (Refer Week 8 Lectures)
- Union-Find data structure in Kruskal's algorithm (refer Week 10 Lectures)
- Efficient Search and retrieval data structures of various kinds (Refer Weeks 5,6,7 lectures)

Don't repeat work!

Do not compute anything more than once (if there is room to store it for reuse)

- Underpins Dynamic Programming strategy
  - Rod-cutting, matrix chain multiplication etc. (Refer Week 4 Lecture)
  - Largest Common Subsequence problem (Refer question 1 in Week 7 Tute sheet)
  - Dijkstra's Algorithm (Refer Week 8 Lectures)
  - Bellman-Ford Algorithm (Refer Week 9 Lectures)

Try well-known problem solving strategies

- Small to Big (Refer Week 2 lectures)
- Divide and Conquer (Refer Weeks 3, 4 lectures)
- Dynamic Programming (Refer Weeks 4, 8, 9 lectures)

Sometimes (!) greed is good

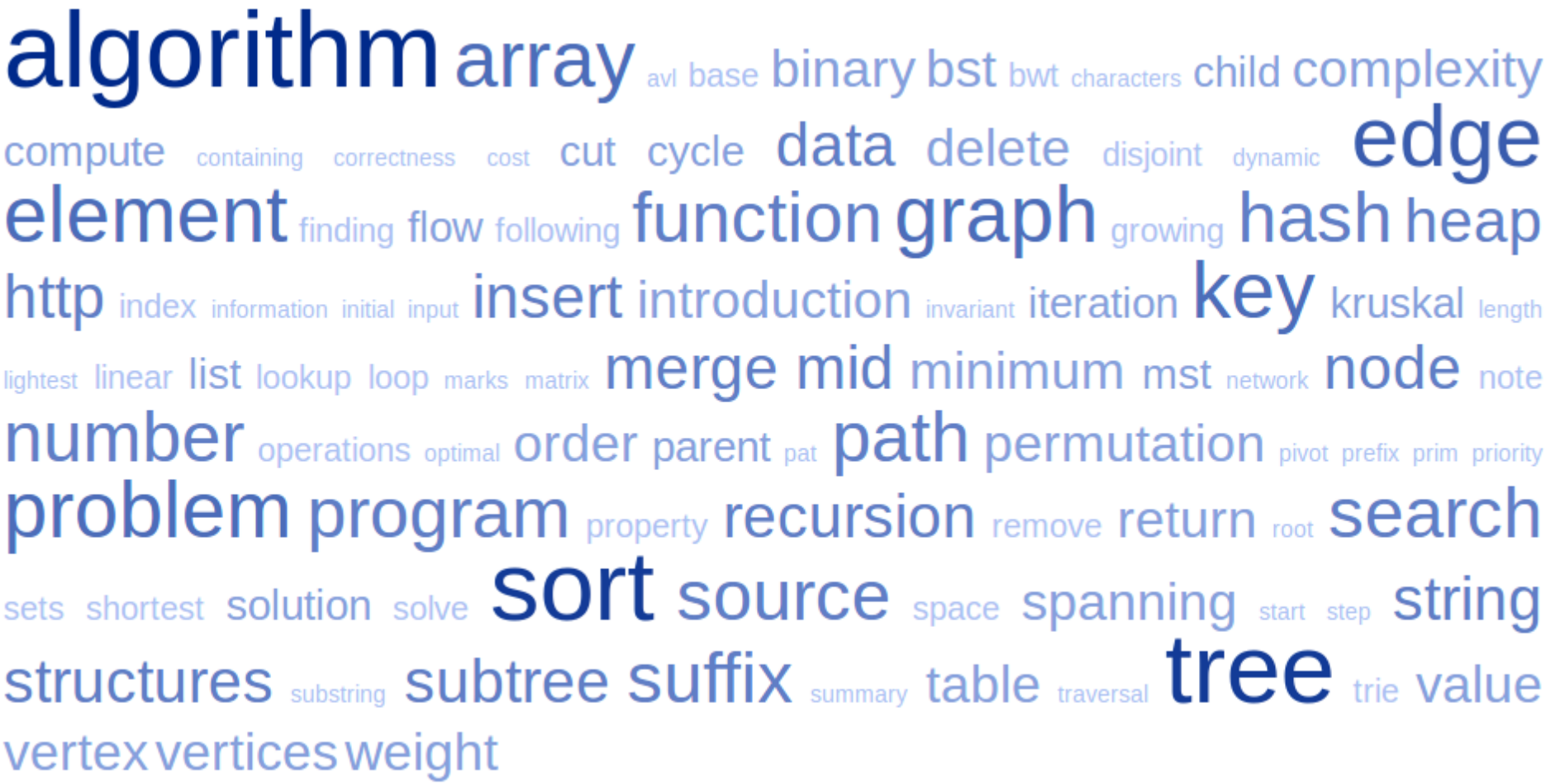
- A **greedy strategy** is to make a "local" choice based on current information (without back-tracking)
- Sometimes gives optimal solution, e.g.
  - Dijkstra's single source shortest paths algorithm (Refer Week 8 Lectures)
  - Minimim Spanning Tree Algorithms – Prim's and Kruskal's (Refer Week 10 lectures) minimum spanning tree algorithm.

Greedy is sometimes a good heuristic!

It gives a "good" solution to a (combinatorial) problem even if not guaranteed optimal.

EPILOGUE – Cloud of FIT2004 words

(combining all LaTeX source files written for your lectures/pracs/tutes)



- The unit has come to an end.
- Hope you have enjoyed this unit as much as I have teaching it – thank you!
- Don't stop your learning with this content – we really scratched the surface!
- Cannot understate the importance of Algorithms & Data Structures in your professional lives that is ahead of you.
- Good bye, keep safe, and be in touch!

BEST OF LUCK FOR YOU EXAM!

```
-=o0o=-
THE END
-=o0o=-
```