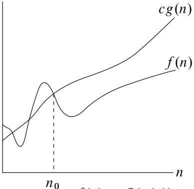
CSc 8530
Parallel Algorithms

Spring 2019

January 17th, 2019

O-notation

- $O(g(n)) = \{f(n) \,:\,$ there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$
- $g(n)$ is an *asymptotic upper bound* for $f(n)$
- If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$
  - **Abuse of notation for convenience**
  - Similarly for the other notations



$f(n) = O(g(n))$

(b)

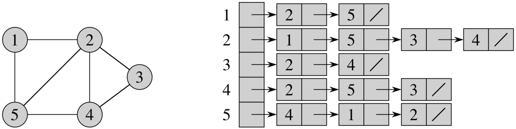
Important classes of algorithms

- Most algorithms you'll encounter in practice are either:
  - **Constant time:** $\Theta(1)$
    - Running time is independent of input size
  - **Logarithmic time:** $\Theta(\log(n))$
    - Running time is proportional to the *number of bits* needed to encode the input
  - **Linear time:** $\Theta(n)$
    - Running time is proportional to the input size
  - **Log-linear time:** $\Theta(n \log(n))$
    - How many times we execute an $\Theta(n)$ operation depends on the input size's number of bits
  - **Polynomial time:** $\Theta(n^p)$
    - Common cases: $\Theta(n)$, $\Theta(n^2)$ (quadratic), $\Theta(n^3)$ (cubic)
    - Running time is proportional to a number of subsets (e.g., pairs for quadratic, triples for cubic, etc.)
  - **Exponential time:** $\Theta(2^{n^p})$
    - Common case: $\Theta(2^n)$
    - Running time *doubles* every time the input size grows by one
    - Practical only for small inputs

Graph definition

- A graph $G = (V, E)$ is defined by two sets:
  - A set of n vertices V (also called nodes)
  - A set of m edges E (also called links)
- All the elements in both V and E are unique (i.e., no repeated values)
- Every edge $e = (u, v) \in E$ is a tuple (i.e., two *ordered* values), such that $u, v \in V$
  - In other words, each edge is defined by its starting and ending vertices
- In general, $m = O(n^2)$ (why?)

Undirected graph

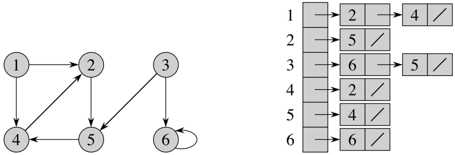


(a) (b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Directed graph



(a) (b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

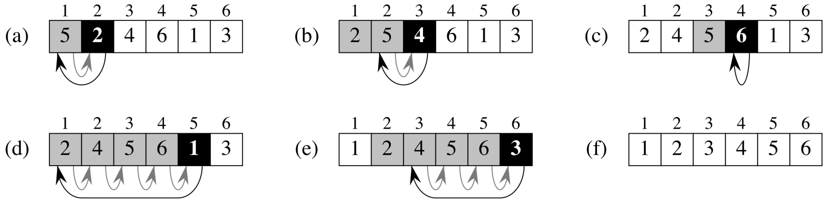
Graph attributes

- We often assign attributes to the vertices and/or edges
- Enables us to represent real-world phenomena (e.g., a map of cities)
- Common cases:
  - An (x, y) vector (usually real values) for each vertex that defines its position on a plane (**planar graphs**)
  - A *weight* (usually real values) for each edge. Weights can represent, among other things, the distance or similarity between neighboring vertices
  - A *weight* for each vertex
  - A string (i.e., name) for each vertex
- More exotic examples can include arbitrary data structures: lists of strings, arrays, even other graphs
- In our pseudocode, we will use $v.d$ to refer to the attribute d for the vertex v

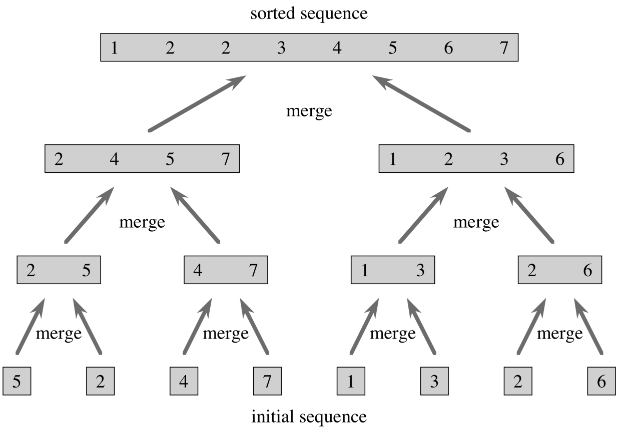
Divide and conquer

- There are many strategies for designing algorithms
- The most basic is **incremental**
    - We iteratively add one element to our partial solution at a time until we have a complete solution
    - Example: insertion sort
- **Divide and conquer** is another common approach
    - **Divide** the problem into smaller instances of the same problem
    - **Conquer** the subproblems by solving them *recursively*
        - **Base case:** if a subproblem is small enough, solve it directly
    - **Combine** the solutions to the subproblems to solve the original problem
    - Example: merge sort

Insertion sort



Merge sort


sorted sequence
1 2 2 3 4 5 6 7
merge
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
initial sequence

Greedy algorithms

- Algorithms for optimizing a value (e.g., the minimum cost of a spanning tree) typically go through a sequence of steps, with a set of choices at each step
- An algorithm is **greedy** if, when faced with a set of possible actions, it always picks the one that looks best at the moment
  - It doesn't factor in how earlier choices influence later ones
- We make a *locally optimal choice* in the hope of getting a *globally optimal solution*

Greedy algorithms

Greedy algorithms work for problems that have:

- **Optimal substructure**
  - *An optimal solution can be constructed efficiently from optimal solutions of its subproblems*
  - Intuitively, the problem can be broken down into separate sub-problems
  - Also applies to **dynamic programming**
  - **Example and counter-example:**
    - Let $p(C_1, C_2)$ be a path between two cities C_1 to C_2
    - Let $D(p(C_1, C_2))$ and $F(p(C_1, C_2))$ be the (minimum) costs of driving and flying, respectively
    - Assume that $p(\text{Atlanta}, \text{Raleigh})$ includes Charlotte
    - Then:

$$D(p(\text{Atlanta}, \text{Raleigh})) = D(p(\text{Atlanta},\text{Charlotte})) + D(p(\text{Charlotte}, \text{Raleigh}))$$

$$F(p(\text{Atlanta}, \text{Raleigh})) \neq F(p(\text{Atlanta},\text{Charlotte})) + F(p(\text{Charlotte}, \text{Raleigh}))$$

Greedy algorithms

Greedy algorithms work for problems that have:

- **Iterative optimality**
    - The current solution is optimal for the subset of the problem observed so far
    - The best current choice may depend on previous choices, but not future ones
    - **Example:**
        - Making change (with US coins) using the fewest number of coins
        - Algorithm: Keep picking the largest denomination, until you go over, then pick the next largest, etc.
        - 36 cents = 1 quarter + 1 dime + 1 penny (3 coins)

Dynamic programming

- **Dynamic programming** (DP) is a powerful optimization technique which breaks a problem into subproblems
  - Similar to divide-and-conquer, but DP caches intermediate results
    - Avoids solving the same subproblem twice
  - Similar to greedy algorithms, but applies to problems where we have to factor in the subsequent cost of an action
    - In the greedy case, we only care about the local, immediate cost
- **Note:** the term "programming" refers to scheduling, not code
- As in the phrases: "Today's reception has been programmed for 5:00pm" or "Get with the program"

Dynamic programming

Dynamic programming works for problems that have:

- **Optimal substructure**
  - *An optimal solution can be constructed efficiently from optimal solutions of its subproblems*
  - Intuitively, the problem can be broken down into separate sub-problems
  - **Example and counter-example:**
    - Let $p(C_1, C_2)$ be a path between two cities C_1 to C_2
    - Let $D(p(C_1, C_2))$ and $F(p(C_1, C_2))$ be the (minimum) costs of driving and flying, respectively
    - Assume that $p(\text{Atlanta}, \text{Raleigh})$ includes Charlotte
    - Then:

$$D(p(\text{Atlanta}, \text{Raleigh})) = D(p(\text{Atlanta,Charlotte})) + D(p(\text{Charlotte}, \text{Raleigh}))$$

$$F(p(\text{Atlanta}, \text{Raleigh})) \neq F(p(\text{Atlanta,Charlotte})) + F(p(\text{Charlotte}, \text{Raleigh}))$$

Dynamic programming

- Dynamic programming works for problems that have:
  - **Overlapping subproblems:** A recursive algorithm has to solve the same subproblems over and over
  - The space of subproblems is *small*:
    - Typically **polynomial** with respect to the input size
    - For general problems, this space has **exponential** size
  - DP stores or caches the solution to each subproblem to avoid having to solve it again
  - Requires a lookup-table-type data structure to keep track of already solved subproblems
  - **Example:** Fibonacci numbers: $f(n) = f(n-1) + f(n-2)$
    - We can save a lot of calculations by solving each $f(n-k)$ only once
    - For $f(5)$ a recursive solution would compute:

$$\begin{aligned} f(5) &= f(4) + f(3) \\ &= (f(3) + f(2)) + (f(2) + f(1)) \\ &= ((f(2) + f(1)) + (f(1) + f(0))) + ((f(1) + f(0)) + f(1)) \\ &= (((f(1) + f(0)) + f(1)) + (f(1) + f(0))) + ((f(1) + f(0)) + f(1)) \end{aligned}$$

Dynamic programming vs. greedy approach

```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  for each vertex u ∈ G.V
      INSERT(Q, u)
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
          if v.d changed
              DECREASE-KEY(Q, v, v.d)
```

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)      // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

- Dijkstra's shortest-path and Prim's minimum-spanning-tree algorithm are virtually identical
- The only difference is in how we update a node's cost:

$$\text{Dijkstra's: } v.d = u.d + w(u, v)$$

$$\text{Prim's: } v.d = w(u, v)$$

- **Historical note:** Dijkstra actually rediscovered and published (1959) Prim's algorithm two years after Prim (1957) (and 29 years after the earliest discover, Vojtěck Jarník (1930))

Dynamic programming vs. greedy approach

- The updates:

$$\text{Dijkstra's: } v.d = u.d + w(u,v)$$
$$\text{Prim's: } v.d = w(u,v)$$

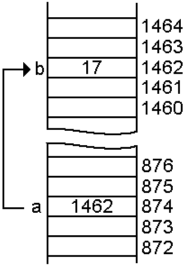
  encapsulate the difference between **dynamic programming** and a **greedy approach**
- **Greedy:** We only have to factor the local, current cost of an item (an edge, in this case)
- **Dynamic programming:** We have to factor the local cost + the best cost assuming we take that action
  - In this case, it is more intuitive to imagine going *backwards* (from v to s)
  - $u.d$ is the minimum cost of going from u to s
  - This is the best we can do, if we choose to first move from v to u

The C programming language

- When C was first introduced in 1972, it was considered a "high-level" language
  - Compared to assembler
  - By modern standards, it is rather low-level
- Has a few gotchas w.r.t. to higher-level languages (e.g., Python, Matlab, or even Java):
  - Manual memory allocation
  - No garbage collection
  - Liberal use of pointers
  - Variables and functions must be declared prior to use (e.g., in a header file)
  - Heavy use of macros (pre-compilation text replacement)
  - No built-in support for string operations, object orientation, etc.

Pointers

- When a variable is initialized, the operating system allocates it a specific memory location
- A pointer is a reference to the memory location of another variable
  - Used to change variables inside a function (reference parameters)
  - Used to remember a particular member of a group (such as an array)
- Pointers are usually much smaller than the data they point to



By Daniel B - de.wikibooks, CC BY-SA 3.0

Pointer arithmetic

- Pointers can be manipulated like other integers
  - Addition, subtraction, multiplication, etc.
- Pointers often point to the start of a data structure (e.g., an array)
- We usually add a constant to a pointer to access different parts of that structure
- Example: `p = p + 1` accesses the adjacent memory location
- **A point of caution:** Trying to access memory locations outside of your program's valid area will result in a segmentation fault

```c
// Variable of type int
int x = 5;

// pnt points to x. & is called a
// derefencing (or address) operator
int *pnt = &x;

// y has the same value as x (5)
// The * operator access the value
// that the pointer points to
int y = *pnt;

// Pointer z points to x
int *z = pnt;

// How are x, y, and z affected?
*pnt = 10;
```

Dynamic memory allocation

- Dynamic memory is allocated during the execution of the program
  - Its specific size is not known in advance
- Dynamic memory handling in C is manual
  - Like in a stick-shift car
- The programmer is responsible for requesting, allocating, and freeing up all dynamic memory
- Generally, through the standard library functions `malloc` and `free`
- A common source of bugs is to access memory before it is allocated or after it has been freed

Dynamic memory allocation

```
// Static memory allocation
int array[10];

// Dynamic memory allocation
// with error checking
int *array = malloc(size*sizeof(int));
if (array == NULL) {
  fprintf(stderr, "malloc failed\n");
  return(-1);
}

// ... use the array

// Deallocate the memory
free(array);
```

Other nuances in C

- C has a **preprocessor** that manipulates the source code before handing it over to the compiler
  - Its main use is for replacing **macros** (basically text substitution)
  - Macros are often used for constants, e.g., `#define PI 3.14`
- Variables must be declared prior to use:
  - Unlike languages such as Python or Matlab
  - e.g. `int x = 0; x = x + 5;`
- Strings are just arrays of characters with a NULL ('\0') value at the end
  - e.g., "cat" is ['c','a','t','\0']
  - No standard methods for comparing, concatenating, etc.
  - Must use array manipulation
- Displaying variable values is awkward
  - You have to manually declare how `printf()` should format each variable
  - e.g., `printf("This is a signed int:  %d",signIntVar)`

Parallel processing

- The goal of parallel processing is to execute a program faster by using multiple processors
- Typically, the processors are all of the same type
- The way the processors are **interconnected** is fundamental
  - Different types of connections lead to radically different types of parallel architectures
- In a **parallel architecture**, the processors are tightly interconnected
  - Usually via some form of shared memory
- In contrast, in a **distributed system** the processors can be heterogeneous and separated geographically

Parallel speedup

- Let P be a computational problem with inputs of size n
- We denote the best-possible sequential (i.e., classic) complexity of P as $T^*(n)$
- Let A be a parallel algorithm that solves P in time $T_p(n)$ using p processors
- Then, the **speedup** achieved by A is:

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

- By construction, $S_p(n) \leq p$
- We would like $S_p(n) \approx p$
  - i.e., each processor should do around $1/p$ of the work of a single one
- In practice, inefficiencies in concurrency, synchronization, communication, etc. reduce the actual speedup

Parallel efficiency

- The **efficiency** of a parallel algorithm A is given by:

$$E_p(n) = \frac{T_1(n)}{pT_p(n)}$$

- $T_1(n)$ is the running time of the parallel algorithm with a single processor
  - Not necessarily equal to $T^*(n)$
- Efficiency measures how much bang for our buck we get per processor
- Ideally, $E_p(n) \approx 1$
- Again, inefficiencies reduce this value in practice

Upper limit on running time

- For a given input size, there is an upper limit $T_\infty(n)$ on how much we can speed up processing with more processors
  - For example, if we are adding two vectors of length n, having more than n processors is of no use
- $T_p(n) \geq T_\infty(n)$, for all p
- Furthermore:

$$E_p(n) \leq \frac{T_1(n)}{pT_\infty(n)}$$

- An algorithm's efficiency degrades quickly once we exceed $T_1(n)/T_\infty$