CPSC 535: Advanced Algorithms

Instructor: Dr. Doina Bein

Terminology

- A (computational) **problem** describes what we want to compute.
  - It is defined by a type of input and a type of output, where any input and output may be represented by a finite digital data structure.
  - The definition of the problem may introduce mathematical variables whose scope is limited to that problem definition.
- Notation:

The <*problem name*> problem is:

input: <definition of input objects>

Output: <definition of output objects>

- Example:

The *minimum* problem is:

input: a list L of n > 0 comparable objects

output: the least element in L

- A *problem instance* (or an *instance*) = a specific, concrete input to a problem.
- Ex: lists [7, 2, 1, 9] is an instance of the minimum problem, as is [1, 2]; empty list [] is not a valid instance of that problem since the problem's input statement requires that n > 0.
- A *process* is a series of actions directed to some end.
- *Algorithm* = an ordered sequence of finite number of steps – each of which can be executed in finite time
- Example of algorithms:
  - Algorithm for adding two integers
  - Algorithm for finding the shortest path in a network
  - Algorithm to search for an article on the web

Remarks

- There can be more than one algorithm to solve a given problem.
- Sorting problem can be solved by using any of the following algorithms
  - Insertion sort
  - Heap sort
  - Quick sort
  - Etc.

Pseudocode

- *Pseudocode* = a human-readable format for communicating algorithms that may include code-like syntax, math notation, and prose.
- Pseudocode is similar to program source code in a language such as C or Java
- The pseudocode for *minimum* problem is:

```
Let min = first element of L (there is one since n > 0)
For each element in L do
    if (min > element) then let min = element
Return min
```

Implementations

- An *implementation* of an algorithm is executable computer code that follows the process defined by the algorithm.
- A computer program describes one particular way of computing a solution to a problem.
- A computer program does not truly solve a problem unless it terminates on all inputs, and produces a correct answer.
- Ex: implementation in C++

```
float min;
int i;
min = L[0];
for (i=1; i < n; i++)
    if (min > L[i])
        min = L[i];
return min;
```

How to measure the quality of an algorithm

- Quality of an algorithm is measured in terms of the resources it consumes when it is executed on a computer
- Example of such resources: memory, communication bandwidth, processor time
- Of interest in this class are the execution (processor) time and the memory need
  - Shorter the execution time, better the quality of the algorithm – *time complexity*
  - Smaller the amount of memory needed for execution, better the quality of the algorithm – *space complexity*

- The *complexity* of an algorithm A, with respect to a specific instance I and resource R, is a non-negative real number representing the amount R that is consumed by A when run on I.
- It would be nice to quantify the ease-of-understanding of an algorithm. But this issue is not discussed in algorithm textbooks.
- It is not useful to measure the execution time of an algorithm in absolute terms (like 10 sec, 30 μ sec, 1 hr, etc.) since the execution time depends on the processor's speed

- We measure the algorithm on a universal model of computation called RAM
- The model of computation and implementation is the **RAM** (random access machine) model:
  - one memory (infinite),
  - one processor and
  - instructions are executed sequentially. The instructions are simple: assignment, addition, multiplication, comparison, etc.
- For parallel computation, the model is called **PRAM** (parallel random-access machine) model

- *Step count* (or *Running Time*) = the number of primitive operations or “steps”. For our RAM model, we assume that executing each instruction (or statement) takes a constant amount of time. The constant values may differ, but they are the same for two similar instructions: e.g. adding two numbers takes the same time if the numbers are integer/floating point, but addition may take less time than subtraction
- Step count (or running time) of an algorithm is the sum of steps (or running times) for each statement executed

Example

- Let us consider the following algorithms:

```
s = 0
for i =1 to n do
    s = s+ i*i
```

```
s = 0
for i =1 to n do
    for j =1 to n do
        s = s+ i*j
```

- Which will require more space?
- Which will require more processor time?

General Rules for Computing the S.C.

- It takes one (1) unit time to do anything simple: addition, multiplication, assignment, comparison, read/write a value
- Consecutive statements add up.
- If/Else: for the fragment

```
If (condition)
    S1
Else
    S2
```

- The s.c. is equal to the number of steps to evaluate the condition plus the maximum between the s.c. of S1 and S2
- For loops: The s.c. of a for-loop is at most the s.c. of the statements inside the for-loop times the number of iterations.
- Nested loops are analyzed inside out.

- Characterize the complexity of algorithms in general so that we can deduce how they might perform when run on unforeseen instances.
- The worst-case complexity of algorithms = the worst (greatest) complexity of that algorithm, over all valid inputs to the algorithm.
  - worst case time complexity
  - worst case space complexity

- *Best case running time*: minimum number of steps. For example, when sorting a sequence of numbers using insertion-sort and the sequence is already sorted
- *Worst-case running time*: maximum number of steps. For example, when sorting a sequence of numbers using insertion-sort and the sequence is reversely sorted
- *Average case running time*: the input is chosen at random, the average number of steps.
- The worst-case running time of an algorithm is an *upper bound* on the running time for any input.
- The average case running time is often as bad as the worst-case running time. E.g. insertion sort has a quadratic time for both, and a linear time for best case.
- Expected running time is defined for randomized algorithms.

- Furthermore, the execution time depends on the *input size*: larger the input size, longer the execution time.
- Input size of an algorithm is the number of characters it takes to code the input
- Examples:
  - Insertion sort: here the input is `n` and the array, thus the input size is 1+the size of the array that is given to sort, which is `n+1`.
  - Shortest path in a network: here the input is `n` (no of vertices), `m` (no of edges), the list of vertices and the list of edges, thus the input size is `1+1+n+m`
  - Sometimes it is necessary to be critical in coming up with input size expression. The input size for primality test (of a number `n`) is `log n` and not `n`, size it takes $\lceil \log n \rceil$ bits to represent `n` in binary

Order of Growth or Rate of Growth

- The time complexity of an algorithm is measured in terms of its growth rate (i.e. order) as a function of input size
- Sometimes computing the exact running time is not worth the effort
- Ex.: Alg. for finding the smallest number in an array a[ ] of size `n`

Algorithm 1:

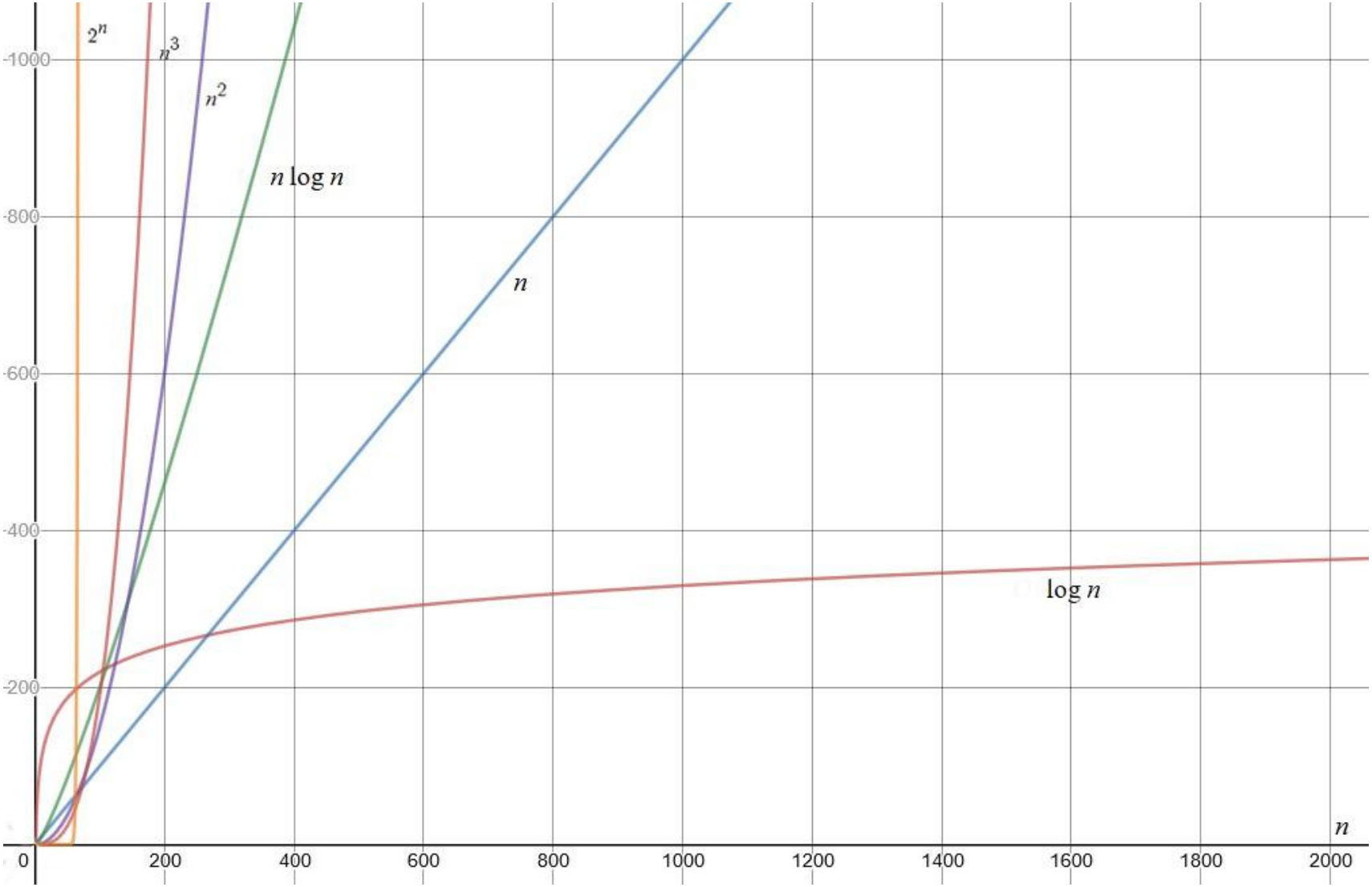
Input: Integer array a[ ] of size n (input size of the above algorithm is `n`)

Output: Smallest entry in a[ ]

```
int small = a[0]
for i=1 to n-1
    if (a[i] < small) then small = a[i]
output small
```

- The execution time is proportional to `n`, the size of the input
- Hence the execution time of Algorithm 1 is linear i.e. `O(n)` ("Big-Oh of n")

- We will define notation O (“Big-Oh”) soon
- There are two simplifying assumptions that we make when analyzing the running time of an algorithm
    1. Constants are ignored
    2. Only the leading term is considered
- A solution that takes constant time has r.t. of O(1).
- An algorithm A is more efficient than another algorithm B if the w.c.r.t of A has a lower order of growth. Examples: n, log(n), n^2, n^2log(n), n^3, n^n


2^n
n^3
n^2
$n \log n$
n
$\log n$
1000
800
600
400
200
0
200
400
600
800
1000
1200
1400
1600
1800
2000
n

- The complexity of some instances is less than or equal the worst-case complexity
- Sometimes small instances do not follow the general trend
- Example: the *minimum selection problem*

  input: a list L of $n \geq 0$ numbers

  output: the minimum value among all elements in L
- Algorithm that solves the problem:

```
def naïve_min (n, L):
  if L is empty, return 0
  else
      Let min = first element of L (there is one since n > 0)
      For each element in L do
          if (min > element) then let min = element
      Return min
```

- Running time:

$$T = \begin{cases} 2 & if\ n = 0 \\ 2(n+1) & if\ n > 0 \end{cases}$$

- This value is a function of n, so we call it:

$$T(n) = \begin{cases} 2 & if\ n = 0 \\ 2(n+1) & if\ n > 0 \end{cases}$$

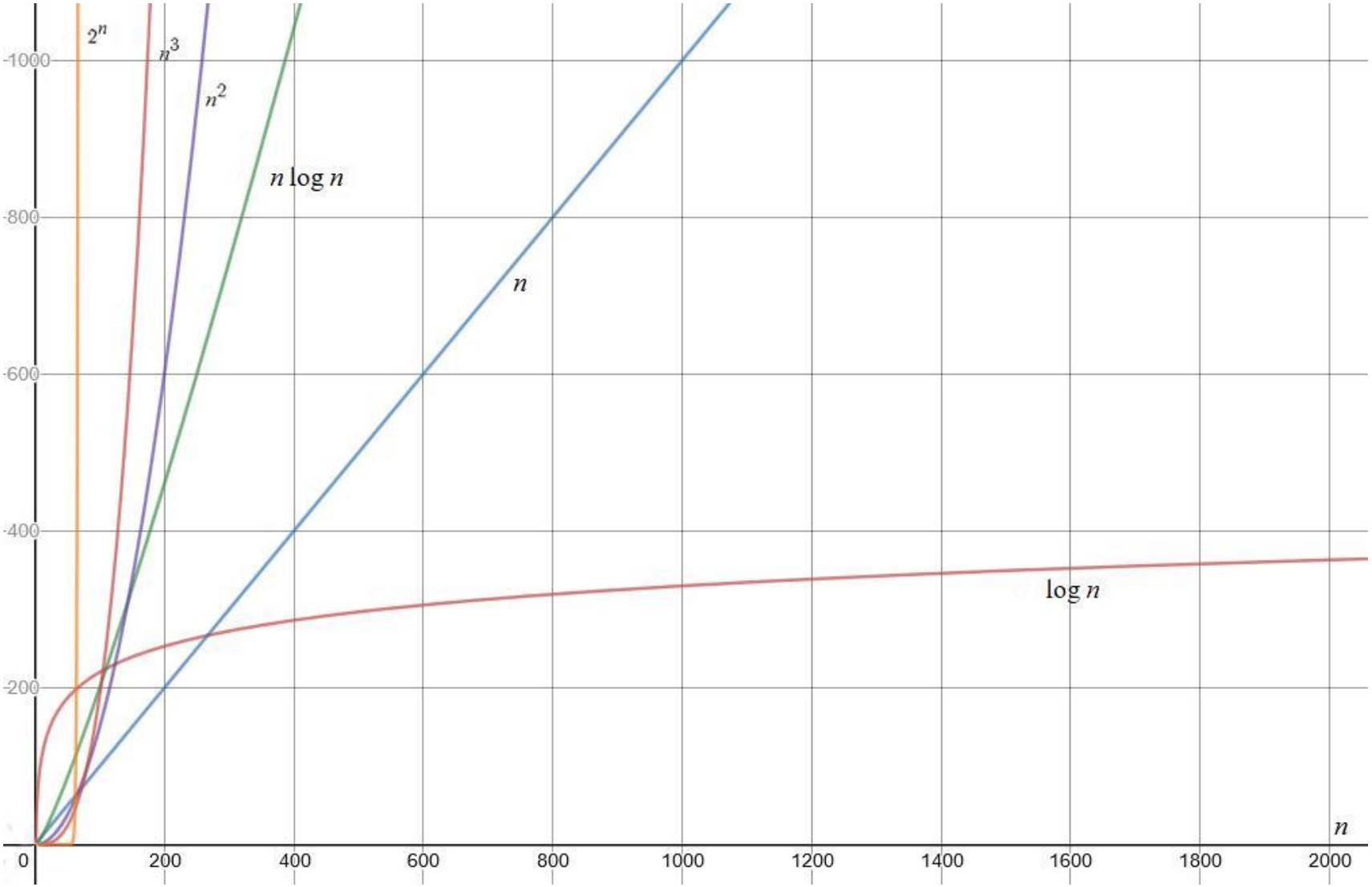
- Note: *When analyzing an algorithm, we assume that there is some threshold for the input size, beyond which the trend is established.*
- Thus we ignore small inputs from our analysis

- The measurement of any resource (time, space) involves a hidden constant:
- When analyzing the time complexity, we compute the number of steps performed by the algorithm on the RAM model
- Each simple instruction takes a precise amount of CPU time on a given computer, and a set of simple instructions (such as an algorithm) will take a multiplicative time
- Takeaway: From the point of algorithms' efficiency, two functions $T_1(n)$ and $T_2(n) = c \cdot T_1(n)$ that differ by only a constant are considered equivalent.

Order of Growth or Rate of Growth

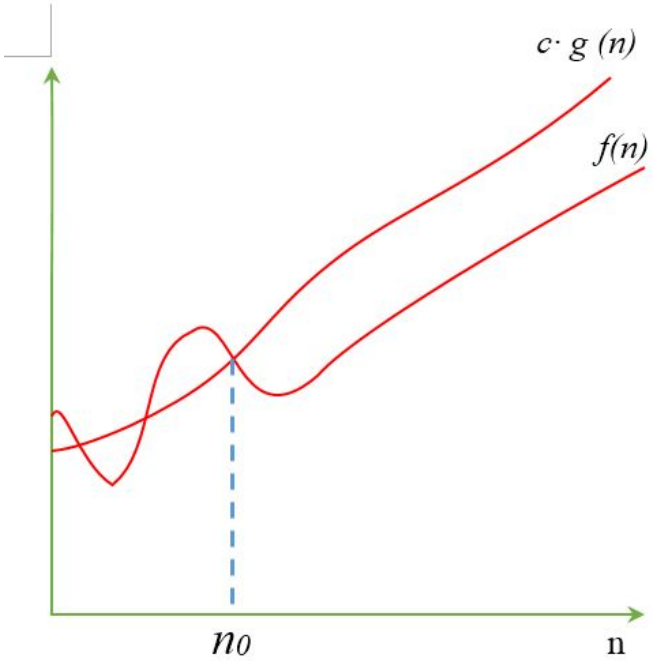
- The time complexity of an algorithm is measured in terms of its *growth rate* (i.e. order) as a function of input size.
- Ex: for the minimum selection problem in a list of size `n`, the execution time is proportional to `n`, the size of the input. Thus the execution time of Algorithm naïve_min is linear i.e. $O(n)$ ("Big-Oh of n"). We will define notation O( ) (Big-Oh) soon.
- We are concerned with how the running time of an algorithm increases with the size of the input *in the limit* (i.e. the size of the input increases without bound).
- For large inputs, only the order of growth is relevant. We say then that we are studying the *asymptotic efficiency of algorithms*.

- An algorithm that is asymptotically more efficient beats the rest for all but very small inputs.
- A solution that takes constant time has r.t. of O(1).
- An algorithm A is more efficient than another algorithm B if the w.c.r.t of A has a lower order of growth. Examples: n, log(n), n^2, n^2log(n), n^3, n^n
- There can be many functions between the consecutive functions shown
- $n^2 \leq n^2 \log \mathrm{n} \leq n^3$
- $\log n \leq \sqrt{n} \leq n$
- So on


2^n
n^3
n^2
$n \log n$
n
$\log n$
1000
800
600
400
200
0
200
400
600
800
1000
1200
1400
1600
1800
2000
n

Big-Oh (O) Notation

- For a given function g(n), a function t(n) is said to be in O(g(n)), denoted t(n) $\in$ O(g(n)), if t(n) is bounded above by some constant multiple of g(n) for all large n
- O(g(n)) denotes the set of functions f(n) for which there exist two positive constants, c and n_0 such that $f(n) \leq c \cdot g(n)$ for every $n \geq n_0$.



- Example 1: $f_1(n) = 10n + 5$, $f_2(n) = n^2$.
  Then $f_1(n) \in O\big(f_2(n)\big)$. Why?

$10n + 5 \leq c \cdot n^2$, n ≥ n_0

Let us choose n_0 = 1 and c = 15:

$10n + 5 \leq 15 \cdot n^2$, n ≥ 1

Or $10n + 5 \leq 10n^2 + 5n^2$, n ≥ 1

This is trivially true.

n^2 is an upper bound of 10n+5

f_2(n) is an upper bound of f_1(n).

- Example 2: Show that $10n + 100 \in O(n)$

Have $f(n) = 10n + 100$ and $g(n) = n$.

$10n + 100 \leq c \cdot n$, $n \geq n_0$

Let us choose $n_0 = 1$ and $c = 110$:

$10n + 100 \leq 110 \cdot n$, $n \geq 1$

$10n + 100 \leq 10n + 100n$, $n \geq 1$

This is trivially true.

- Example 3: Show that $5n^3 + 100n \log n \in O(n^3)$

$5n^3 + 100n \log n \leq 105n^3$, n ≥ 1

or, $5n^3 + 100n \log n \leq 5n^3 + 100n^3$, n ≥ 1

(This is trivially true.)

- Remark:
  $n^2 \in O(n^3), n^2 \in O(n^3 \log n), n^2 \in O(n^4)$ etc.

But $n^2 = O(n^2)$ is the tight upper bound.

- Example 4: Prove that every polynomial $p(n) = a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} + \cdots + a_0$, with $a_k > 0$ belongs to $O(n^k)$.

Let SM be the sum of absolute values of all a_k, a_{k-1}, …, a_0.

Then we can write

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} + \cdots + a_0$$

$$\leq |a_k| n^k + |a_{k-1}| n^k + |a_{k-2}| n^k + \cdots + |a_m|$$

$$\leq (|a_k| + |a_{k-1}| + \cdots + |a_0|) n^k, \text{ n>1}$$

$$\leq SM \cdot n^k, \text{ n} \geq 1$$

Thus $p(n) = O(n^k)$

- Example 5: Show that the statement $3n^2 + 2n + 10 \in O(n)$ is false.
  We can also write that $3n^2 + 2n + 10 \notin O(n)$.

Assume $3n^2 + 2n + 10 \in O(n)$.

Thus there exist c > 0 and $n_0 \geq 0$ such that $3n^2 + 2n + 10 \leq c \cdot n$, for all $n \geq n_0$

Or, $3n + 2 + \frac{10}{n} \leq c$, for all $n \geq n_0$

Remember that c is a constant. Also $3n + 2 + \frac{10}{n}$ gets arbitrarily very large for large values of n, thus it cannot be upper bounded by a constant. Contradiction.

- To show that a certain function f(n) belongs to O(g(n)) we either use the definition by finding the values for the constants c and n_0 or we use the following theorem:

Theorem: *If f and g are univariate complexity functions,* $g(n) > 0$, $\lim_{n \to \infty} \frac{f(n)}{g(n)} = L$, *and L is non-negative and constant with respect to n then* $f(n) \in O\big(f(n)\big)$.

- Example 6: Let α and β be real positive numbers such that $0 < \alpha < \beta$. Show that $n^{\alpha} \in O(n^{\beta})$.
- To solve this problem we use the theorem:

Let $\beta - \alpha = \epsilon$, ϵ is positive.

We take to the limit $\lim_{n\to\infty} \frac{n^{\alpha}}{n^{\beta}} = \lim_{n\to\infty} \frac{n^{\alpha}}{n^{\alpha+\epsilon}}$=

$\lim_{n\to\infty} \frac{1}{n^{\epsilon}} = 0$. L=0 is a non-negative constant with respect to n thus $n^{\alpha} = O(n^{\beta})$.

Using limits, show that:

- Show that $10n + 5 \in O(n^2)$
- Show that $10n + 100 \in O(n)$
- Show that $5n^3 + 100n \log n \in O(n^3)$
- Prove that every polynomial $p(n) = a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} + \cdots + a_0$, with $a_k > 0$ belongs to $O(n^k)$.
- $17 \in O(18)$.

Remember that:

- We can drop additive constants
- We can drop multiplicative constants
- We can drop dominated terms (and keep only the dominating term); due to the

Lemma: For any complexity functions $f_0(n)$ *and* $f_1(n)$, $O\big(f_0(n) + f_1(n)\big) = O\left(max\big(f_0(n), f_1(n)\big)\right)$

Thus $O(n^2 + 2^n) = O(2^n)$

- We can drop floor and ceiling operators

Remember that:

(these can be proven by using calculus)

- Log functions grow more slowly than any power of n functions, including fractional power.
  i.e. $\log n \in O(n^{\alpha}), \alpha > 0.$
  In fact $\log n \in o(n^{\alpha}), \alpha > 0$
- Power of n grows more slowly than exponential functions such as 2^n

  i.e. $n^k \in O(2^n)$

  In fact $n^k \in o(2^n)$
- $\log^k n \in O(n)$. In fact, $\log^k n \in o(n)$

The top efficiency classes

Notation	Name	Example
O(1)	constant	Evaluating one statement
O(log n)	logarithmic	Searching a balanced search tree
O(n)	linear	For loop
O(n log n)	“n-log-n”	Fast sorting algorithms such as heap sort or merge sort
$O(n^2)$	quadratic	Two nested for loops
$O(n^3)$	cubic	Three nested for loops
$O(2^n)$	exponential	All subsets of an n-element set
O(n!)	factorial	All permutations of an n-element sequence

- Upper bound is expressed as O(g(n)) (“Big-Oh”) and lower bound is expressed as Ω (g(n)) (“Big-Omega”)

Solving a problem

- A problem has an input and an output
- A solution to the problem can be an algorithm that takes the input and produces the output after a finite number of steps
- We compare solutions by comparing algorithms, in terms of time and space complexity
- The upper bound of a decidable problem is the time complexity of some particular algorithm that solves that problem:
  - Problem X with input *n* is solved by algorithm $solve_X$ that has the time complexity *f(n)* and there is no algorithm with a lower asymptotic time complexity, so we say that problem X has the upper bound O(*f(n)*)
- A decidable problem has also a lower bound

Lower bound for a problem

- A decidable problem has a lower bound
- An algorithm DOES NOT have a lower bound
- The lower bound for a particular problem is the best possible time complexity of any algorithm solving the problem
- The notation for lower bound is Ω

$$\Omega(f(n)) = \{g(n) \mid \text{there exists some constants } c > 0 \text{ and } t \geq 0 \text{ such that } g(n) \geq c \cdot f(n) \text{ whenever } n \geq t\}$$

- If we know the lower bound for a given problem and succeed in designing an algorithm whose complexity matches that lower bound, then we are effectively done.
- If a problem has the lower bound of is Ω (t(n)) then any correct algorithm solving that problem will have the time complexity O(t(n)) or slower

Sorting problem

- The sorting problem's lower bound is Ω(n log n), which means that every algorithm that solves the sorting problem has time complexity O(n log n) or slower.
- We will never see a sorting algorithm faster than O(n log n); O(n), for example, is out of the question
- We have seen sorting algorithms with O(n log n) and O(n^2) time complexities, which are compatible with this lower bound.
- Even slower sorts, which take O(n^3) time, say, are also conceivable and consistent with this lower bound.
- When a problem's upper and lower bounds match, we say that the problem has a *tight bound*.

Tight bound

- When problem X has an upper bound of O(f(n)) and matching lower bound Ω(f(n)), we say that X has a tight bound of Θ(f(n)).
- When there is a gap between the best known lower and upper bounds for a problem, there is an open question as to which bound can be improved on.
- We emphasize that upper bounds and lower bounds are properties of problems, not algorithms.

Big-Omega (Ω) Notation

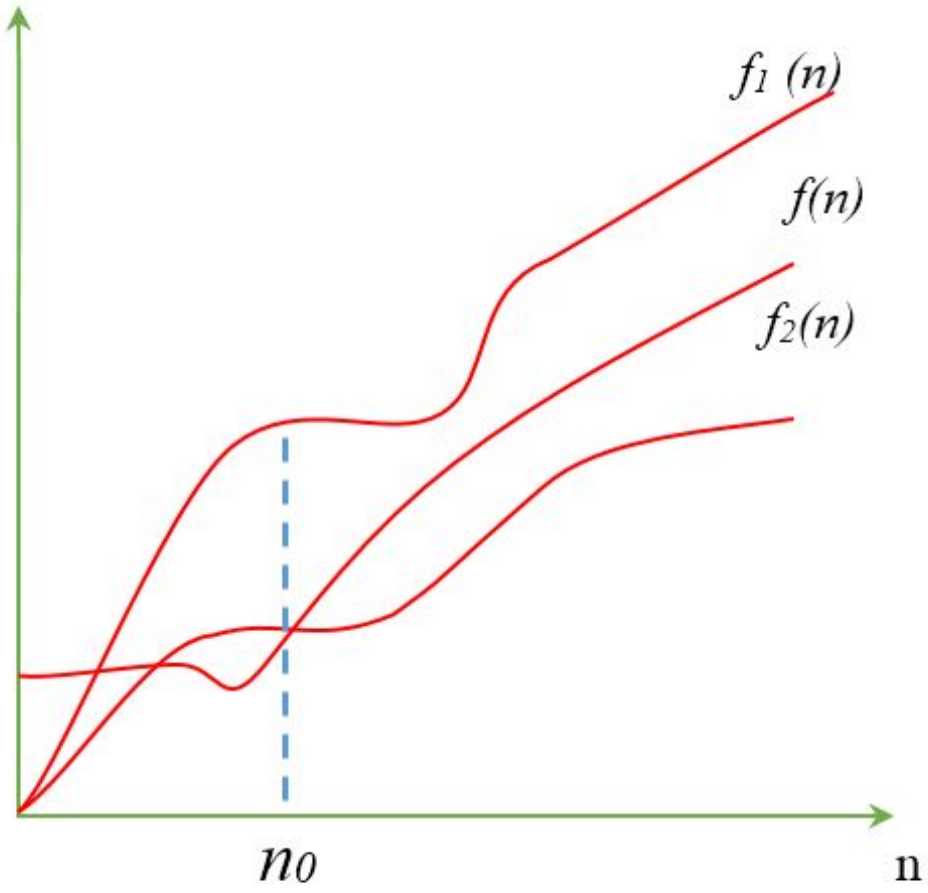
- For a given function g(n), a function t(n) is said to be in Ω(g(n)), denoted $t(n) \in \Omega(g(n))$, if t(n) is bounded below by some constant multiple of g(n) for all large n
- Ω(g(n)) denotes the set of functions f(n) for which there exist two positive constants, c and n_0 such that $f(n) \geq c \cdot g(n)$ for every $n \geq n_0$.

Big-Theta (Θ) Notation

- For a given function g(n), a function t(n) is said to be in Θ(g(n)), denoted $t(n) \in \Theta(g(n))$, if t(n) is bounded both above and below by some constant multiple of g(n) for all large n
- Θ(g(n)) denotes the set of functions f(n) for which there exist three positive constants, c_1, c_2, and n_0 such that $c_1 \cdot g(n) \geq f(n) \geq c_2 \cdot g(n)$ for every $n \geq n_0$.

Understanding the upper bound & lower bound of a function

- Consider three functions



- For large values of n,
  (i) $f_1(n)$ is larger than f(n), and
  (ii) $f_2(n)$ is smaller than f(n)

  We say that $f_1(n)$ is an *upper bound* for f(n)

  We say that $f_2(n)$ is a *lower bound* for f(n)

Proving the lower bound for sorting

- An upper bound, such as “there exists an algorithm that solves the sorting problem in O(n log n) time,” can be proven by example. To prove this claim, we only need to show one example of an algorithm that solves the sorting problem in O(n log n) time.
- A lower bound, we need to prove that, for every algorithm that every has been or could be conceived, that algorithm’s time complexity is O(n log n) or slower

Trivial lower bounds in general

- A correct algorithm must, at a minimum, initialize the data structures that hold its output:
- ***Claim****: If the space complexity of the output for problem X is O(f(n)), then X has a lower bound of* $\Omega(f(n))$.
- For sorting problem, the output is a data structure with space complexity O(n), therefore, there is a trivial lower bound of Ω(n) time for the sorting problem
- For the sequential search problem, from this Claim we get that the lower bound is Ω(1), a constant. Not helpful.
- So we need other properties.

Trivial lower bounds in general (cont.)

- A correct algorithm must interact with some amount of input data:
- *Claim: If every correct algorithm for problem X must observe O(f(n)) data, then X has a lower bound of* Ω *(f(n)).*
- Any correct sequential search algorithm must be capable of examining all n elements, in O(n) time, so we have a lower bound of Ω(n) for sequential search.

Binary Search Problem

- Some algorithms are clever about examining only a small subset of their input which is relevant.
- In the binary search problem, the classical binary search algorithm examines only O(log n) input elements, rules out half of the input at every step, and thereby ignore almost all of the input, n-O(log n) elements entirely.
- The binary search algorithm has worst-case time complexity of O(log n), which establishes an upper bound of O(log n) time for that problem
- *Claim: The binary search problem has a tight bound of Θ(log n).*

Tight bound for sorting

- Every correct sort must be capable of starting from a (likely-unordered) vector V and "learning" which permutation will re-order V into non-decreasing order.
- Any sorting algorithm must make a series of comparisons of the type "compare if a < b for two elements a, b ∈ V", and uses the outcome of these comparisons to guide the sorting process
- Nearly all practical sorting algorithms are comparison-based although radix sort is one notable exception.
- ***Theorem****: The sorting problem has a lower bound of* $\Omega(n \log n)$ *that applies to all comparison-based sorting algorithms.*

- **Proof**: Let A be an arbitrary correct decision-based sorting algorithm that re-orders the elements of V into a sorted vector S in non-decreasing order.
- We assume that all elements of V are distinct and there exists exactly one valid permutation p.
- V has n elements, so there are n! candidate permutations that might possibly be a valid p.
- Let a and b be the first two elements of V that are compared by algorithm A, and we have that $a \neq b$
- So either $a < b$ or $a > b$.
  - When $a < b$, we know that a must precede b in S, so S has the form S =...a...b..... Of the n! permutations of V, exactly half fit this pattern where a comes before b.
  - When $a > b$,S has the form S =...b...a... , and only the other half of the permutations fit this pattern.

- The first comparison (between a and b) gives to the algorithm A enough information to decrease the size of the set of candidate permutations by half.
- The second comparison of elements could provide A enough information to halve the candidate set again to n!/4 permutations, the third comparison could halve the candidate set to n!/8 permutations, and so on.
- From the master theorem and analysis of binary search, a decrease-by-half decision process narrows a field of k candidates down to one result after only O(log k) decisions. Therefore A makes O(log(n!)) comparisons.

$$\log(n!) \in \Omega(n \log n).$$

Rules

- If $T_1(n) = O(f(n))$ and $T_2(n)=O(g(n))$ then
  - $T_1(n) + T_2(n) = O(f(n)+g(n))$ or $O(\max(f(n),g(n)))$ – sequential statements
  - $T_1(n) * T_2(n) = O(f(n)*g(n))$ – for loops
- If $T(n)$ is a polynomial of degree k then $T(n)=\Theta(n^k)$
- $\text{Log}^k n=O(n)$ for any constant k
- **L'Hôpital rule**: if $\lim_{n\to\infty} f(n) = \infty$ and $\lim_{n\to\infty} g(n) = \infty$ then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f(n)'}{g(n)'}$ where $f(n)'$ and $g(n)'$ are the derivatives of f(n) and g(n).

Using limits when comparing order of growth

- When comparing two functions, f(n) and g(n):
  - If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then f(n) has a smaller asymptotic growth rate than g(n), i.e. f(n) ∈ O (g(n))
  - If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \neq 0$ then f(n) has the same asymptotic growth rate as g(n), i.e. f(n) ∈ Θ (g(n))
  - If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ then f(n) has a larger asymptotic growth rate than g(n), i.e. f(n) ∈ Ω (g(n))

Approximation of the factorial function

- The factorial is defined as $n! = \begin{cases} 1 & if\ n = 0 \\ n \cdot (n-1) & if\ n > 0 \end{cases}$

It is trivial to see that $n! \leq n^n$.

Stirling's approximation (or Stirling's formula) is an approximation for factorials. It is a very powerful approximation, leading to accurate results for even small values of n:

$$n! \cong \sqrt{2\pi\, n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

General Computational Problems

- A problem has an input and an output
- A solution to the problem can be an algorithm that takes the input and produces the output after a finite number of steps
- Def: Given an alphabet Σ, a *computational problem* is a function mapping strings defined on Σ to set of strings on Σ,
- Computational problem ≠ decision problem
- Any particular input to a problem is called an *instance* of the problem.

Categories

- All the computational problems discussed in this course fit into the definition, but in real-world they are stated as questions or tasks, not functions
- Based on their task, we divide them into:
  - Function problems
  - Search problems
  - Optimization problems
  - Threshold problems
- Each computational problem may have an associated *decision problem*

Function Problems

- A *function problem* is characterized by a function f(I) that returns a single string as output, where I is the input string, and is defined as follows:

  Given input string I, compute f(I) (or return "no" if f(I) is not defined).

- The output of a *function problem* is always a singleton string (we consider "no" to be a singleton string, too).

Example 1: Multiplication à la russe

- Read more on Internet about it, also called “Ethiopian multiplication” or “Peasant multiplication” (https://en.wikipedia.org/wiki/Ancient_Egyptian_multiplication , https://www.wikihow.com/Multiply-Using-the-Russian-Peasant-Method)
- The way most people learn to multiply large numbers looks something like this:

```
   86
 x 57
------
   602
+ 4300
------
 4902
```

- This "long multiplication" is quick and relatively simple.
- For the Ethiopian / à la russe / Russian peasant algorithm, one does not need multiplication; one only needs to double numbers, cut them in half, and add them up, but it will take longer to get the result.

How it works?

- **Follow the rules**:
- Write each number at the head of a column, it does not matter which column; let's consider that the first number is in the left column and the second number in the right column.
- Halve the number in the left column and double the number in the right column as follows:
  - If the number in the left column is odd, divide it by 2 and drop the remainder.
  - If the number in the left column is even, cross out that entire row.
- Keep halving, doubling, and crossing out rows until the number in the left column is 1.
- Add up the remaining (uncrossed) numbers in the right column.
- The total is the product of your original numbers.

Why it works?

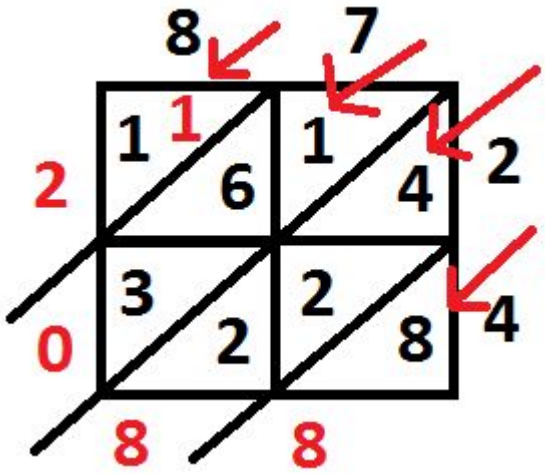
- Let m and n be the two numbers

- If m is even, then the first row will contain
  m n
- And the second row will contain
  m/2 2*n
- Note that m*n=(m/2)*(2*n).
- The first row will be crossed out and the result will be computed based on second (possibly) and the subsequent rows.

- If m is odd, then the first row will contain
  m n
- And the second row will contain
  (m-1)/2 2*n
- Note that m*n = m+((m-1)/2)*(2*n).
- The first row will be kept and the result will be computed based on first, second (possibly) and the subsequent rows.

Example 2: Lattice Multiplication

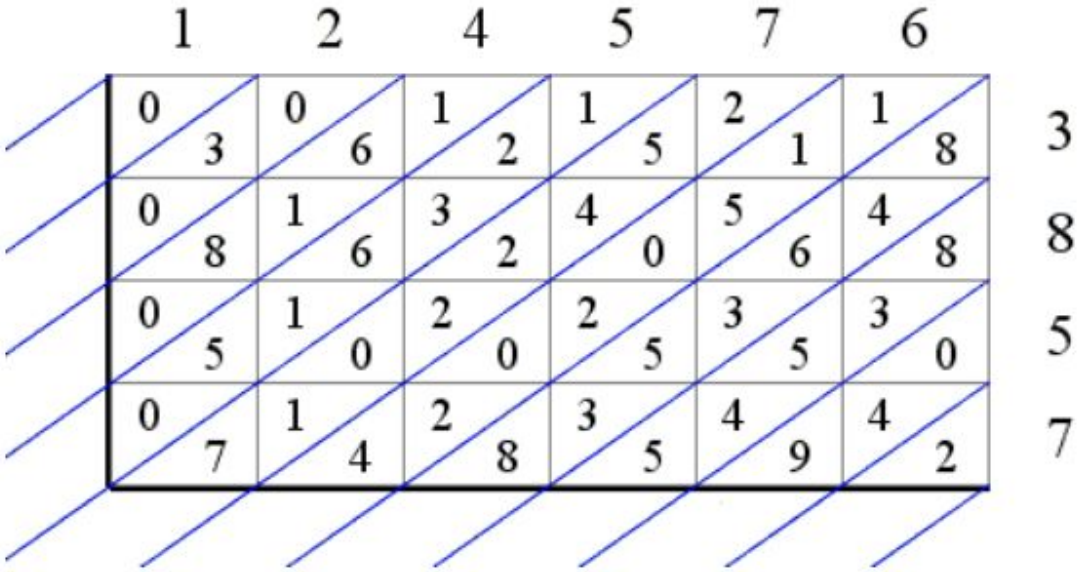
- Read more on Internet about it, known under many names, most common “lattice multiplication” (https://en.wikipedia.org/wiki/Lattice_multiplication)



How it works?

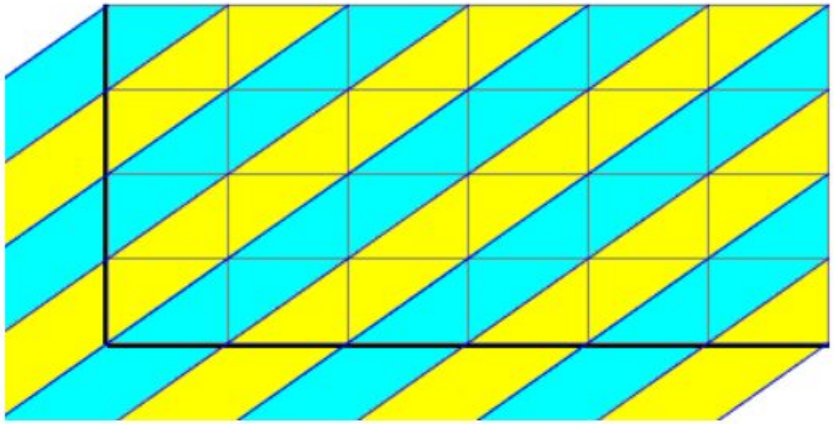
(https://www.cut-the-knot.org/Curriculum/Arithmetic/LatticeMultiplication.shtml)

- The number with the smallest number of digits goes on the row and the number with the largest number of digits goes on the column, one digit per cell.
- A grid is drawn up, and each cell is split diagonally.
- The two multiplicands of the product to be calculated are written along the top and right side of the lattice, as below:

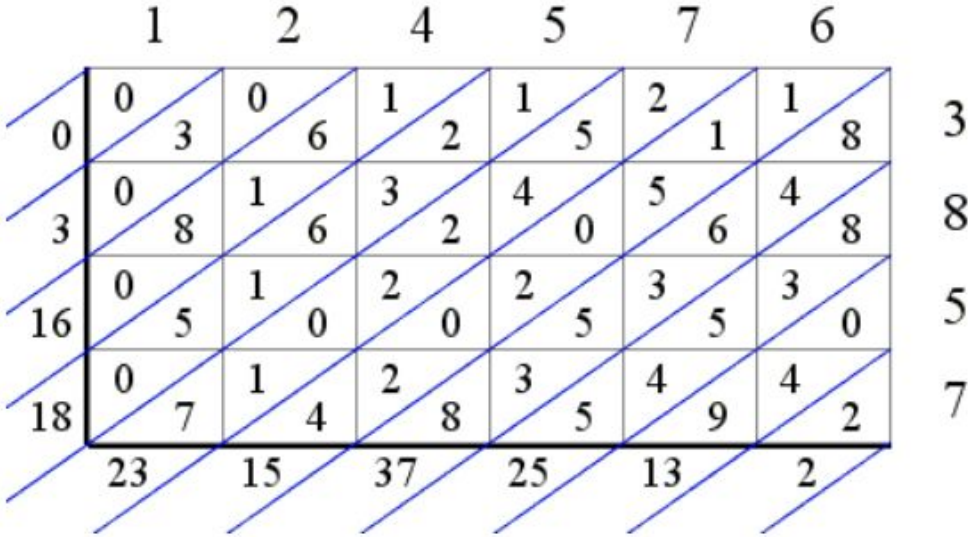


(contd.)

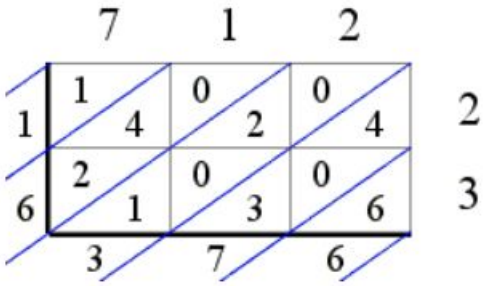
- Note that the diagonals split the lattice into (diagonal) bands:



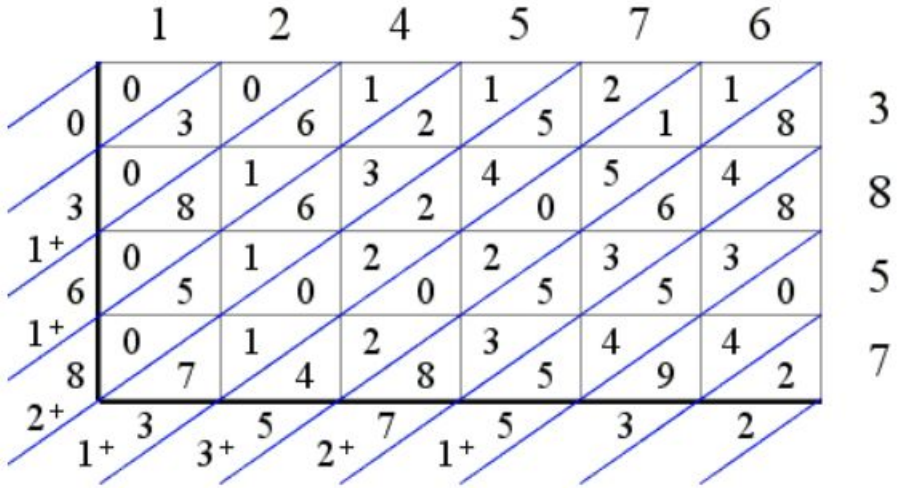
- The algorithm requires to compute the sums of all the digits in a band and place the result next to the lattice, to the left or below the bottom, as the case may be.
- The product 124576×3857 leads to the following diagram:



- The sums are thought of as being ordered around the lattice, first from the top to the bottom and then left to right.
- If all the sums are single digit numbers, the product of the two multiplicands can be read right away by following the sums around the lattice *counterclockwise*, so 712×23 = 16376:



- The 2-digit sums cause a complication which is resolved by carrying their first digit to the previous band (counterclockwise) and adding it to what remains there of the sum placed there previously.
- The product 124576×3857 leads to the following diagram:



- The third intermediate sum 2 + 8 = 10 takes two digits leading to an additional carry of 1 which is added to the preceding sum 1 + 6 making it 1 + 6 + 1 = 8.
- Once all the sums are single digit numbers, the product of the two multiplicands can be read right away by following the sums around the lattice counterclockwise.
- We obtain 124576×3857 = 480489632.

Search Problems

- A predicate is a function that returns true or false.
- A search problem is characterized by a predicate Q(I,S) where I is the input string and S is some string, as follows:

  Find a string S such that Q(I,S) is true (or return "no" if no such S exists).

Optimization Problems

- A *numerical function* is a function that returns numbers, usually integer-valued.
- An *optimization problem* is characterized by a numerical function V(I,S) where I is the input string and S is some string, as follows:

  Find a string S such that V(I,S) is minimized (or maximized).

Threshold Problems

- A *threshold problem* is characterized by a numerical function V(I,S) and a numerical threshold K, where I is the input string, K is an input numerical value, and S is some string, and is defined as follows:

  Given I and K, find a string S such that V(I,S) ≤ K (or ≥, or =).

- Another variant asks a decision version of the same question:

  Given I and K, does there exist an S such that V(I,S) ≤ K (or ≥, or =)?

Decision Problems

- A *decision problem* is a function problem with exactly two possible solutions, “yes” or “no”, i.e. it is characterized by a function f(I) that returns “yes” or “no”, where I is the input string, and is defined as follows:

  Given the string I, returns “yes” if f(I) exists or “no” if f(I) is not defined.

- Are rarely used in practical applications, but they are relatively easy to state and prove theorems about it.
- Most general computational problems have a closely related decision problem that captures the same fundamental idea, and vice-versa.

Strategies to Design Algorithms

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

- Why?
  - novel scholarly research
  - industrial software development (algorithm engineering)
  - technical interviews
  - class exercises
  - exam questions
- Follow a checklist (next)

Checklist

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

1. Understand the problem definition
2. Baseline algorithm for comparison
3. Goal setting: improve on the baseline how?
4. Design a more sophisticated algorithm
5. Inspiration (if necessary) from patterns, bottleneck in the baseline algorithm, other algorithms
6. Analyze your solution; goal met? trade-offs?

1. Understand

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

- Read the problem definition (input/output) carefully and critically.
- Make sure that you understand the data types involved, constraints on them, technical terms, and story of the problem.
- Write out a straightforward input and output. Try additional examples involving non-obvious solutions.
- What are the corner cases of input? Can the input be empty?
- When is the solution obvious, and when is it not?
- What is a use case for this algorithm in practical software?
- Does this problem resemble other problems you know about?
- If any of this is unclear, **ask questions**.

Example Problem: Word Find [1]

Problem description: A popular diversion in the United States, “word find” (or “word search”) puzzles ask the player to find each of a given set of words in a square table filled with single letters. A word can read horizontally (left or right), vertically (up or down), or along a 45 degree diagonal (in any of the four directions) formed by consecutively adjacent cells of the table; it may wrap around the table’s boundaries, but it must read in the same direction with no zigzagging. The same cell of the table may be used in different words, but, in a given word, the same cell may be used no more than once. Write an algorithm for solving this puzzle.

Quiz:

- Data types of input and output, constraints on these data types (if any)
- Problem instance, i.e. example of input and output
- Are there any special cases, so called “Corner cases”?
- Use case, i.e. where solving this problem can be useful
- Similar to anything familiar?

[1] Introduction to the Design and Analysis of Algorithms. By Anand Levitin, ex. 10, page 107

2. Baseline

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

Now that you understand the problem, quickly identify at least one algorithm that solves it.

- (research context) literature review: CLRS, Wikipedia, Google Scholar
- sketch a naive algorithm: brute force, exhaustive search, for loops
- What is the efficiency of your baseline algorithm?
- Baseline alg. almost always exists
  - in our contexts, problems are almost always in NP
  - → exponential-time exhaustive search algorithm
- In the unlikely event no alg. seems to exist, consider whether the problem is provably undecidable.

Stop after the Baseline?

Consider: baseline algorithm is a working, but is possibly a very inefficient solution to your problem.

- might be acceptable if efficiency is low priority
- shows technical interviewer that you can communicate and think about algorithms
- earns partial credit on class assignments
- time/labor management

Pareto principle (80/20 rule): in many settings, spending 20% of maximum effort achieves 80% of maximum benefit.

3. Goal

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

What about the baseline do we want to improve?

- better time efficiency (most common goal)
- better space efficiency
- avoid randomization
- avoid amortization
- extra feature e.g. sort is in-place
- simplicity; elegance; easier to implement/understand
- better constant factors

(non-research contexts): goal may be dictated to you

4. Design

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

Now, design an algorithm, hopefully achieving your goal.

Start with verbal discussion; informal sketches; snippets of pseudocode, prose, equations

As ideas develop and become more concrete, move to pseudocode.

Finally write complete, clear pseudocode for the entire algorithm.

“Devil is in the details:" resist urge to avoid a tricky part, often that is the key to meeting your goal.

5. Inspiration

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

Step 4. Design...

1. might come naturally, devise an algorithm effortlessly
2. more likely: not immediately apparent how to proceed; stuck at first

Inspiration:

- more likely a learned skill
- requires practice, effort, experience
- point of class exercises
- why algorithm design is an in-demand skill

Sources of Inspiration: algorithm design patterns

Patterns taught in CPSC 335:

- greedy
- divide-and-conquer
- randomization
- reduction to another algorithm (sorting) or data structure (hash table, search tree, heap, etc.)
- dynamic programming

Run through the list; can you think of how to use any of these patterns to solve the problem at hand?

If a pattern doesn't seem to work, why is that? Might give a hint about what would work instead.

Sources of Inspiration: Identify Bottleneck

Review the analysis of your baseline algorithm (either from the literature, or what you just devised).

Identify the dominating term in the efficiency analysis.

Trace that term backwards to a part of the baseline algorithm; probably a particular loop or data structure operation.

How can we do less work there?

- ✔ preprocessing (ex. maximum subarray)
- ✔ use an appropriate data structure (ex. heap sort)
- ✔ reuse work instead of repeating work (ex. Dijkstra's alg.)
- ✔ dynamic programming

Sources of Inspiration: Other Algorithms

One reason we study specific algorithms in detail is to learn about clever “tricks" other designers have used, that we might be able to use in novel circumstances.

Examples:

- define invariants to keep yourself organized (selection sort)
- break down problem into simpler phases (heap sort)
- use pointers so you can change many paths w/ one assignment; redirection (search trees)
- use an array instead of pointers (heapsort, open addressing)
- master theorem insights to refactor work out of dominating term (selection)
- when almost all candidate solutions are clearly right/wrong,
- randomize (quicksort, universal hashing)
- compute word-at-a-time (hashing, radix sort)

6. Analyze

https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/01-problem-solving.pdf

Finally analyze your algorithm.

Does it meet your goal? (time efficiency, space efficiency, randomization, etc.)

If yes, obviously done.

If not,

- Is your algorithm still an improvement over your baseline?
- Is more effort justified? (Pareto principle.)
- If yes, go back to prior steps.

Importance of Sorting

https://users.cs.duke.edu/~reif/courses/alglectures/skiena.lectures/lecture4.1.pdf

Sorting problem

- The sorting problem's lower bound is Ω(n log n), which means that every algorithm that solves the sorting problem has time complexity O(n log n) or slower.
- ***Theorem****: The sorting problem has a lower bound of Ω(n log n) that applies to all comparison-based sorting algorithms.*
- But there are sorting algorithms with O(n) time complexity. How?
- If we sort without comparisons, i.e. without comparing entire elements with each other, but maybe parts of it, for example digits of it
- Counting sort, radix sort, and bucket sort, all have O(n)

(Chapter 8 of CLRS)

Counting Sort

(https://users.cs.duke.edu/~reif/courses/alglectures/demleis.lectures/lec5.pdf)

- No comparisons between elements
- Input: A[1..n], an array with n elements to be sorted such that the elements are drawn from $A[j] \in \{1, 2, \ldots, k\}$
  Output: B[1..b] (which is the sorted A)
  Auxiliary storage: C[1..k]
- Algorithm:

```
for i=1 to k do C[i] = 0
for j=1 to n do
  C[A[j]] = C[A[j]] + 1          // C[i] = |{ key == i }|
for i=2 to k do
  C[i]=C[i] + C[i-1]         // C[i] = |{ key ≤ i }|
for j=n downto 1 do
  B[C[A[j]]] = A[j]
  C[A[j]] = C[A[j]] - 1
```

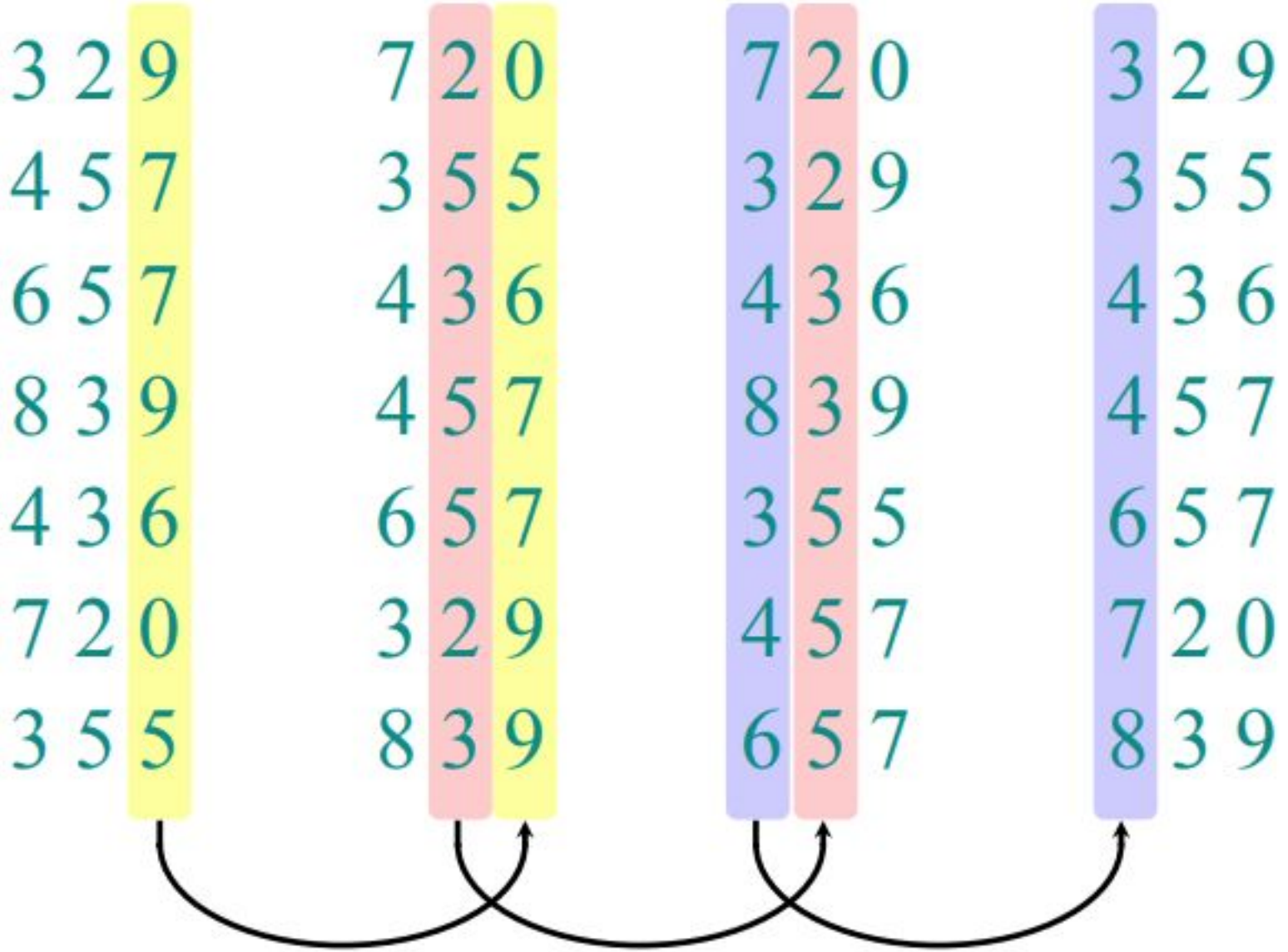
More on Counting Sort

- See the slides 13 through 27 at https://users.cs.duke.edu/~reif/courses/alglectures/demleis.lectures/lec5.pdf
- Time complexity: O(n+k); if k = O(n) then counting sort takes O(n). Why?
- Because counting sort is not a comparison-based sort. No comparison takes place between elements.
- Counting sort is a *stable* sort: it preserves the input order among equal elements. What other sorting algorithm is stable? Answer: mergesort.

Radix Sort

- Origin: Herman Hollerith's card sorting machine for US census in 1890.
- It is a digit by digit sort, sorting on *least significant digit first* using some auxiliary stable sort; Hollerith's idea was to sort starting with the most significant digit which is bad.
- Two elements are not compared by their whole value, but by their digits
- Read the slides 32 through 35 at https://users.cs.duke.edu/~reif/courses/alglectures/demleis.lectures/lec5.pdf
- Radix sort has $O(d \cdot n)$ time where the values to be sorted are in the range $0..n^d-1$ (i.e. at most d digits)

Example


3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
7 2 0
3 5 5
7 2 0
3 5 5
4 3 6
4 5 7
6 5 7
3 2 9
8 3 9
7 2 0
3 2 9
4 3 6
8 3 9
3 5 5
4 5 7
6 5 7
3 2 9
3 5 5
4 3 6
4 5 7
6 5 7
7 2 0
8 3 9

Bucket Sort

- Assumes that the input is drawn from a uniform distribution over the interval [0,1), i.e. the input is generated by a random process that distributes elements uniformly and independently over the interval [0,1)
- **Input**: n numbers from the interval [0,1), uniformly distributed

  Output: sorted n values
- Bucket sort steps:

1. Divide the interval [0,1) into n equal-sized subintervals (or buckets)
2. Distribute each input value into the corresponding bucket
3. Simply sort the numbers in each bucket
4. Go through the buckets in order, listing the elements in each

Example

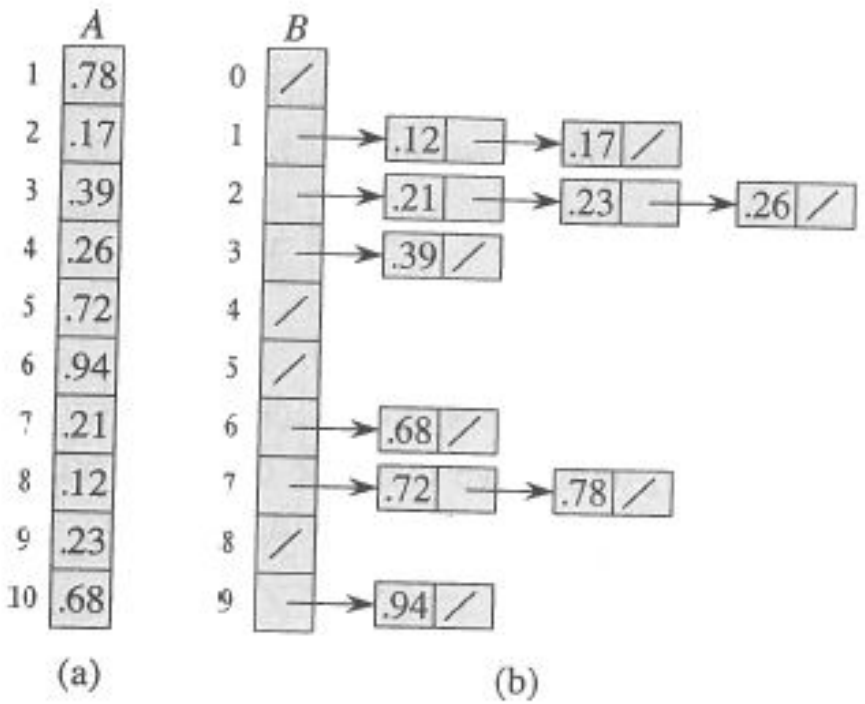


Figure 8.4 The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 8 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i + 1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Bucket Sort (contd.)

- We expect few numbers (i.e. a constant number) in each subinterval
- We use insertion sort to sort the elements in each subinterval, which is O(1) expected
- Average-case is $\Theta(n)$ as long as the sum of the squares of the bucket sizes is linear in the total number of elements

Fundamental Data Structures

- Short review of them
- The open content book available at http://opendatastructures.org/ covers them in detail
- PDF version: http://opendatastructures.org/ods-python.pdf
- HTML version: http://opendatastructures.org/ods-python/
- The book is geared towards a Python implementation of these data structures, but the fundamentals are the same, independent of the implementation
- Other websites about teaching introductory computer science algorithms, including searching, sorting, recursion, and graph theory:

https://www.khanacademy.org/computing/computer-science/algorithms

Abstract Data Types (ADT)

- An abstract data type (ADT) is a set of objects that are related to each other together with a set of operations:
  - Description of the objects that compose the data type
  - Description of the relationships between the individual objects
  - Description of the operations to be performed on the objects
- An ADT is a *mathematical abstraction*: nowhere in its definition is any mention of *how* the operations are implemented
- Familiar examples of data types: integers, reals, booleans, arrays.
- Examples of ADT: lists, sets, graphs, trees, along with their operations

ADT versus Its Implementation

- The familiar examples of data types are built-in in programming language (you do not need to worry about their implementation; it is already done)
- An ADT is language-independent and must be implemented in an appropriate computer language
- Difference between an abstract data type (ADT) and the implementation: how each ADT functions and not how is implemented
- An ADT may be implemented in several ways using the same programming language
- If an implementation is done correctly, the program that use them does not necessarily need to know which implementation was used => *information hiding*

Operations on Data Types

- Data type Integers: addition (+), subtraction (-), multiplication (*), integer division (/), comparisons (<, <=, >, >=, ==, !=) etc.
- Data type Reals: addition (+), subtraction (-), multiplication (*), division (/), comparisons (<, <=, >, >=, ==, !=) etc.
- Data Type Booleans: and (&&), or (||), not (!).
- Data Type Array: fetch and store using []

Array ADT

- An ***array*** is a sequence indexed by a system of integer coordinates
- The system of indexes is used to access elements and to permit alteration of individual elements.
- The elements can be accessed directly in Θ(1) time (constant time)
- The type array is built in any high level programming language, but we can still define our own type array and implement it in a different way
- Two integers `m` and `n`, `m:n` denotes all integers in the range `m..n`
- In a `k`–dimensional array `A`, the index i_j in the dimension `j` ranges between $m_j : n_j$: $A[i_1, i_2, \ldots i_k]$ is the element of array `A` with index $(i_1, i_2, \ldots i_k)$
- To store a `k`-dimensional array $A[m_1 : n_1,\ m_2 : n_2, \ldots,\ m_k : n_k]$ in memory the obvious way is to store it using consecutive memory cells

- A storage allocation function $\mathtt{Loc}(\mathtt{A}[\mathtt{i}_1,\mathtt{i}_2,\ldots\mathtt{i}_k])$ is a mapping function that assigns a distinct memory address to each distinct array index $(\mathtt{i}_1,\mathtt{i}_2,\ldots\mathtt{i}_k)$ of array $\mathtt{A}$
- Each element of the array can occupy L memory cells; for example, an integer needs four (4) memory cells
- For a linear (1-dimensional) array X(0:6), the first element X[0] is stored at location X, the second element X[1] is stored at location X+L, the i-th element is stored at location $X + L \cdot i$
- For multi-dimensional arrays that whose elements have the same size (the same type), the most popular storage allocation methods: lexicographic (or row-major order) and column-major order

Lexicographic (Row-Major) Storage Allocation

- A lexicographic ordering of the integer k-tuples $(i_1, i_2, \cdots, i_k)$ in the set $m_1 : n_1 \times m_2 : n_2 \times \cdots m_k : n_k$ is defined as $(a_1, a_2, \cdots, a_k) < (b_1, b_2, \cdots, b_k)$ iff there is some j in the range $1 \leq j \leq k$ such that $a_i = b_i$ for all $1 \leq i <$ j and $a_j < b_j$
- Example: the indices of a 2-dimensional array A[0:2,0:3] in lexicographic order are
  $(0,0) < (0,1) < (0,2) < (0,3) < (1,0) < (1,1) < (1,2) < (1,3) < (2,0) < (2,1) < (2,2) < (2,3)$
- Example: the indices of a 3-dimensional array A[0:2,0:3,0:2] in lexicographic order are
  (0,0,0) < (0,0,1) < (0,0,2) < (0,1,0) < (0,1,1) < (0,1,2) < (0,2,0) < (0,2,1) < (0,2,2) < (0,3,0) < (0,3,1) < (0,3,2) < (1,0,0) < (1,0,1) < (1,0,2) < (1,1,0) < (1,1,1) < (1,1,2) < (1,2,0) < (1,2,1) < (1,2,2) < (1,3,0) < (1,3,1) < (1,3,2) < (2,0,0) < (2,0,1) < (2,0,2) < (2,1,0) < (2,1,1) < (2,1,2) < (2,2,0) < (2,2,1) < (2,2,2) < (2,3,0) < (2,3,1) < (2,3,2)

- If we display the elements of A[0:2,0:3] as a rectangular array (see below) then lexicographic storage (see right) means that first row 0 of A is stored in locations 0:3, then row 1 of A is stored in locations 4:7, and finally row 2 of A is stored in locations 8:11. Thus the name *row-major order*.

Location	Array element
0	A[0,0]
1	A[0,1]
2	A[0,2]
3	A[0,3]
4	A[1,0]
5	A[1,1]
6	A[1,2]
7	A[1,3]
8	A[2,0]
9	A[2,1]
10	A[2,2]
11	A[2,3]

A[0,0]	A[0,1]	A[0,2]	A[0,3]
A[1,0]	A[1,1]	A[1,2]	A[1,3]
A[2,0]	A[2,1]	A[2,2]	A[2,3]

- Given an arbitrary array `A[2:7, 3:9]`, `Loc(A[4,5])` can be calculated as follows:

$$Loc(A[4,5]) = Loc(A[2,3]) + row\ 2 + row\ 3 + L \cdot (5 - 3 + 1)$$

where means $row\ 2$ the number of memory cells to store the entire row 2 which is $L \cdot (9 - 3 + 1)$. Thus

$$Loc(A[4,5]) = Loc(A[2,3]) + (4 - 2) \cdot L \cdot (9 - 3 + 1) + L \cdot (5 - 3 + 1)$$

- In general, given a 2-dimensional array $A[m_1 : n_1, m_2 : n_2\ ]$, the formula for row-major order storage allocation of

$$Loc(A[i_1, i_2\ ]) = Loc(A[m_1, m_2])$$
$$+L \cdot \big((i_1 - m_1) \cdot (n_2 - m_2 + 1) + (i_2 - m_2)\big)$$

- Or $Loc(A[i_1, i_2\ ]) = Loc(A[m_1, m_2\ ]) + c_0 + c_1 \cdot i_1 + c_2 \cdot i_2$ where

$$c_2 = L$$
$$c_1 = L \cdot (n_2 - m_2 + 1)$$
$$c_0 = -L \cdot m_1 \cdot (n_2 - m_2 + 1) - L \cdot (m_2 - 1)$$

- General formula for row-major order storage allocation of an array $\mathrm{A}[m_1 : n_1, m_2 : n_2, \cdots, m_k : n_k]$

$$Loc(A[i_1, i_2, \cdots, i_k]) = Loc(A[m_1, m_2, \cdots, m_k]) + c_0 + c_1 \cdot i_1 + c_2 \cdot i_2 + \cdots + c_k \cdot i_k$$

for suitable constants c_i $(1 \leq i \leq k)$

- Row-major order allocation is efficient:
  - Traversal of elements in a direction parallel to any arbitrary axis of coordinate system is easy and can usually be implemented using machine index registers
  - Access to an arbitrary element can be computed using the formula given above

Column-major Storage Allocation

- If we display the elements of A[0:2,0:3] as a rectangular array (see below) then *column-major order* storage (see right) means that first column 0 of A is stored in locations 0:2, then column 1 of A is stored in locations 3:5, then column 2 of A is stored in locations 6:8, and finally column 3 of A is stored in locations 9:11.

A[0,0]	A[0,1]	A[0,2]	A[0,3]
A[1,0]	A[1,1]	A[1,2]	A[1,3]
A[2,0]	A[2,1]	A[2,2]	A[2,3]

Location	Array element
0	A[0,0]
1	A[1,0]
2	A[2,0]
3	A[0,1]
4	A[1,1]
5	A[2,1]
6	A[0,2]
7	A[1,2]
8	A[2,2]
9	A[0,3]
10	A[1,3]
11	A[2,3]

- Column-major order is different from row-major (lexicographic) order since, for instance, A[2,0] appears earlier in column-major order than A[0,1] even though A[0,1] comes before A[2,0] in row-major (lexicographic) order
- In general, given a 2-dimensional array $\mathrm{A}[m_1 : n_1, m_2 : n_2\ ]$, the formula for column-major order storage allocation of

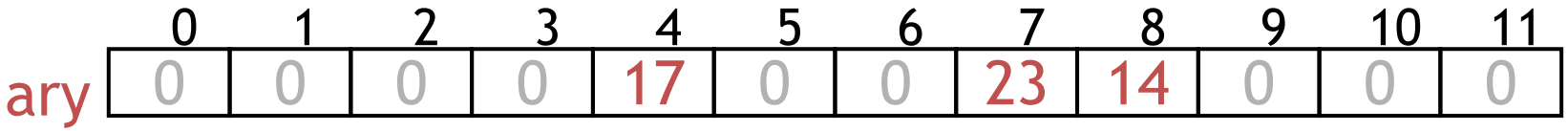
$$Loc(A[i_1, i_2\ ]) = Loc(A[m_1, m_2])$$
$$+L \cdot \big((i_2 - m_2) \cdot (n_1 - m_1 + 1) + (i_1 - m_1)\big)$$

- Two-dimensional arrays are stored in column-major order in FORTRAN

Sparse Array ADT

- A *sparse array* is simply an array most of whose entries are nil (or 0 or some other default value)
- Example: Suppose you wanted a 2-dimensional array of course grades, whose rows are CSUF students and whose columns are courses
- There are about 45,000 students (a fact)
- There are about 3,000 courses (assumption, I could not find the number of courses)
- This array would have about 135,000,000 entries
- Since most students take fewer than 3000 courses, there will be a lot of empty spaces in this array
- This is a big array, even by modern standards
- There are ways to represent sparse arrays efficiently

- We will start with sparse one-dimensional arrays, which are simpler



- We could represent it as a linked list

- We need to get values from the array by operation *fetch*

```
<element type> fetch(int index)
```

which searches a linked list for a given index

- And to store values in the array by operation *store*

```
void store(int index, <element type> value)
```

which searches a linked list for a given index and inserts the element in the list

Data Structures

Data Structure: method for storing, organizing data

- Data members e.g. head pointer, tail pointer
- Invariant(s) defining how the structure must be organized to remain valid, e.g. head points to first node, tail points to last node
- Defined operations, each operation is an algorithm that operates on the structure.

Dynamic Sets

- Sets that can change over time are called *dynamic*
- A *dictionary* is a dynamic set that supports the following operations: insert elements into, delete elements from, test membership in a set.
- Other operations can be supported
- Each element in a set is represented by an *object* whose fields can be examined and manipulated
- Some types of dynamic sets assume that one of the object's field is an identifying *key* field (aka key)
- Some types of dynamic sets assumes that the keys are drawn from a totally ordered set (e.g. real numbers, all words in a dictionary under the usually alphabetic ordering)

Operations on Dynamic Sets

- The operations can be grouped into two types:
  - Queries: return info about the set, an element in the set, or a group of elements in the set
  - Modifying operations: change the set
- Typical operations for a set S:
  - `Search(S, k)`: given `S` and a key value `k`, return a pointer `x` to an element in `S` whose key is `k` (`key[x]=k`) or `NIL`, if `S` does not contain such an element (query)
  - `Insert(S,x)`: augment `S` with the element pointed by `x`, assuming that all the fields in `x` have been initialized (modifying op.)
  - `Delete(S,x)`: given a pointer `x` to an element of `S`, remove `x` from `S` (modifying op.). Note: `x` is a pointer and not a key value. (modifying op.)
  - `Minimum(S,x)`: given a totally ordered set `S`, return the element of `S` with the smallest key value (query)

Operations on Dynamic Sets

- `Maximum(S,x)`: given a totally ordered set `S`, return the element of `S` with the largest key value (query)
- `Successor(S,x)`: given an element `x` whose key is from a totally ordered set `S`, return the next larger element in `S`, or `NIL` if `x` is the maximum element (query)
- `Predecessor(S,x)`: given an element `x` whose key is from a totally ordered set `S`, return the next smaller element in `S`, or `NIL` if `x` is the minimum element (query)

- Successor and Predecessor can be extended to sets with non-distinct keys

Priority Queue

- A priority queue is a list (an ADT) that maintains `S` elements, each with an associated value called a *key*
- A max-priority queue supports the following operations:
  - `Init(S)` : Initialize the priority queue to be empty.
  - `Is_empty(S)` : Test to see whether the priority queue is empty.
  - `Insert(S,x)` : inserts the elements x into the set S. It can be written $S=S \cup \{x\}$
  - `Maximum(S)` : returns the element of S with the largest key
  - `Extract-max(S)`: removes and returns the element of S with the largest key
  - `Increase-key(S,x,k)` : increases the value of element `x`'s key to the new value `k`, which is assumed to be at least as large as `x`'s current key value

Priority Queue

- Stacks, queue, heaps (min heap and max heap) are examples of a priority queue:
  - In a stack, the priority item is the most recently inserted item (called the *top*).
  - In a queue, the priority item is the least recently inserted item (insert at *front*; peek and delete from *rear*)
  - In a heap, the priority item is the item with the minimum (or maximum) key. The key could be anything.

Example: The items of a PQ are unfulfilled obligations.

- The Bill Payer problem: You get bills in the mail from time to time, and you have to pay them. But you don't pay bills the moment you receive them: instead, you place them on your desk, where they are unfulfilled obligations. When you decide to pay a bill, you take one from your desk and pay it.
- *The stack strategy*: When you get a bill, you place it on the top of your pile of bills. When you decide to pay a bill, you pay the one on top, i.e., the one you got most recently.
- *The queue strategy*: The bill you pay is the one that's been on your desk the longest. One way to do this is to insert every new bill at the bottom (rear) of the pile, and take from the top (front) of the pile.
- *The heap strategy*: Every bill has a due date. When you decide to pay, you pay the bill that has the earliest due date. How would you implement this in practice?

Stack (aka LIFO queue) operation	Worst Case
Create an empty stack	O(1)
Push an element into the stack	O(1)
Return the top element of the stack	O(1)
Pop and return the top of the stack	O(1)
Test whether the stack is empty	O(1)

Can I do the following operations?

- Return the bottom of the stack. Answer: NO
- Test whether an element is in the stack. Answer:
    - YES: I can only test the top element
    - NO: I cannot check the entire stack

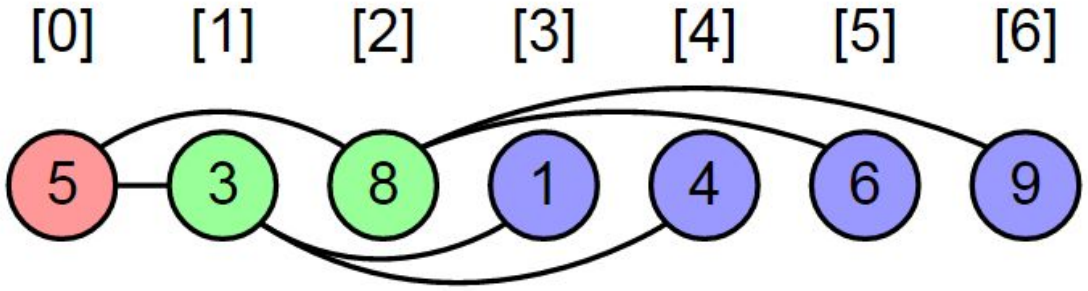
Queue (FIFO queue) operation	Worst Case
Create an empty queue	O(1)
Push an element into the queue	O(1)
Return the rear of the queue	O(1)
Pop the queue	O(1)
Pop and return the rear of the queue	O(1)
Test whether the queue is empty	O(1)

Can I do the following operations?

- Return the front of the queue. Answer: NO. (Technically, I can do it since I have access to the front of the queue, but it violates the property of a queue ADT, when only the element at the rear of the queue can be returned.)
- Test whether an element is in the queue. Answer:
    - YES: I can only test the rear element
    - NO: I cannot check the entire queue

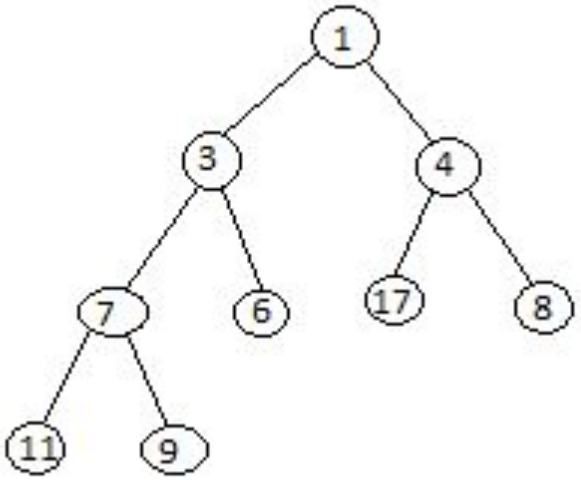
Heaps

- A (binary) heap is a *complete* binary tree and the nodes in the tree follow a *heap-order property*.
- A *complete* binary tree is completely filled except maybe the lowest level, which is filled from left to right.
- A complete binary tree can be represented as an array : array `A` has `length[A]` elements but only the indices `0.. heapsize[A]` are used for the heap.



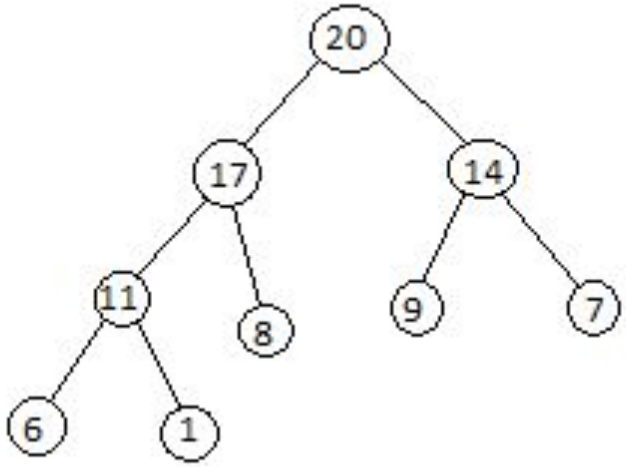
- The root of the tree is stored at `A[0]`
- Given the index `i` of a node:
  - `A[(i-1)/2]` stores the parent of `i`, `Parent(i)`
  - `A[2*i+1]` stores `i`'s left child `Left(i)` and
  - `A[2*i+2]` stores `i`'s right child `Right(i)`

- *Heap-order property*:
  - A tree follows the *heap-order* property if it follows either the max-heap or the min-heap property.
  - A tree follows the *max-heap* order property if the value stored at every node `x` is ≤ the value stored at the parent of `x`, except the root which has no parent: `A[Parent(i)] ≥ A[i]` or `A[(i-1)/2] ≥ A[i]`
  - A tree follows the *min-heap* property if the value stored at every node `x` is ≥ the value stored at the parent of `x`, except the root which has no parent: `A[Parent(i)] ≤ A[i]` or `A[(i-1)/2] ≤ A[i]`



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

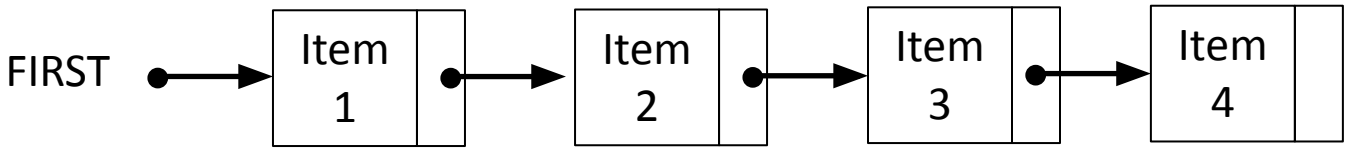
Figures taken from
http://www.studytonight.com/data-structures/heap-sort

- Maintaining the Heap Property for max-heaps:
  - If an element `A[i]` violates the max-heap property w.r.t. its left child i.e. `A[i] < A[2*i+1]` and/or w.r.t its right child `A[i] < A[2*i+2]` but the subtrees rooted at `Left(i)` and `Right(i)` are max-heaps, then `A[i]` needs to be push down the tree

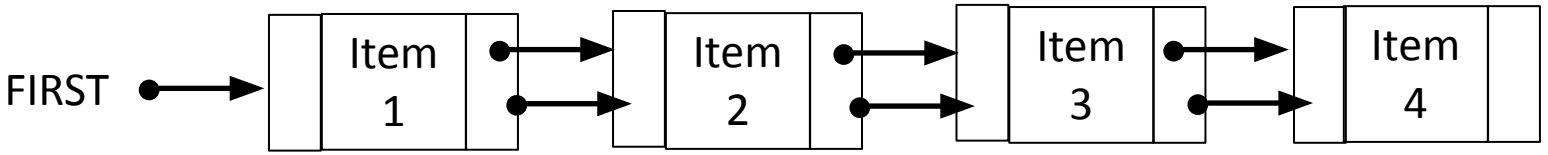
Min-heap operation	Worst Case
Create an empty heap	O(1)
Convert an unsorted list into a heap in-place	O(n)
Push an element into the heap	O(log n)
Find the minimum element into the heap	O(1)
Pop the minimum element out of the heap & restructure heap	O(log n)
Return the top of the heap (aka the minimum element)	O(1)
Pop, return the top of the heap & restructure heap	O(log n)
Test whether the heap is empty	O(1)

Linked Lists

- A list is a finite sequence of items drawn from some set
- The simple linked lists are singly or doubly:
- A *singly linked list* is a list in which each element points to its successor; a list item stores an element and a pointer to its successor



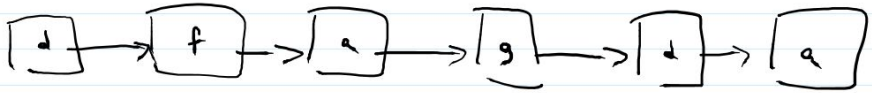
- In a *doubly linked list*, each element points to its successor and to its predecessor; a list item stores an element and two pointers, one to its successor and one to its predecessor



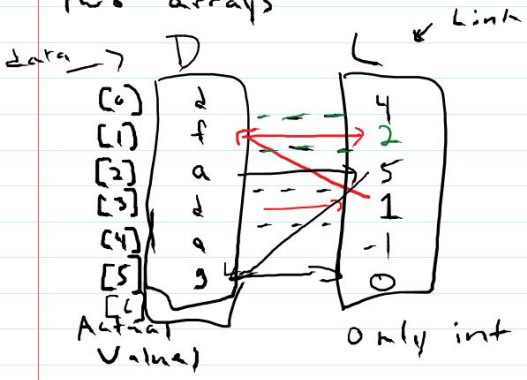
Implementing the list using arrays

- Given a list of length n, two arrays D and L, of length n can be used to implement the list and the list operations
- The array D will contain the elements in the list
- The array L will contain the index of the subsequent elements
- For a position i, $\mathrm{D}[i]$ will store some element in the list and $\mathrm{L}[i]$ will store the index in the array D of the element following $\mathrm{D}[i]$ in the list
- Examples

List: d, f, a, g, d, a



Two arrays



head = 3 (the first element is stored in index 3)

D[head] = first element in the list

L[head] = index of the next element

(D[i], L[i]) = a node in the linked list

head = 3

L[3] = 1 → next element is stored at index 1

L[1] = 2 L[5] = 0 L[4] = -1

L[2] = 5 L[0] = 4

List: m, p, g, a, d, b

	D	L
[0]	a	3
[1]	b	-1
[2]	g	0
[3]	d	1
[4]	p	2
[5]	m	4

head = 5

L[head] = index of D of next element

L[L[head]] = index of next next

L[L[L[head]]] = index in D of the 4th element

Any type of data that can be stored in the list can be stored in D

Only ints can be stored in L

List operation	Worst Case
Create an empty list	O(1)
Test whether the list is empty	O(1)
Insert an element in the list at the head	O(1)
Insert an element anywhere besides the head	O(n)
Insert an element at the end when the tail is maintained	
Insert an element at the end when the tail is not maintained	O(n)
Search for an element based on its value (key)	O(n)
Delete an element in the list based on its value (key)	O(n)

The delete operation is based on which operation?
(Answer: search)

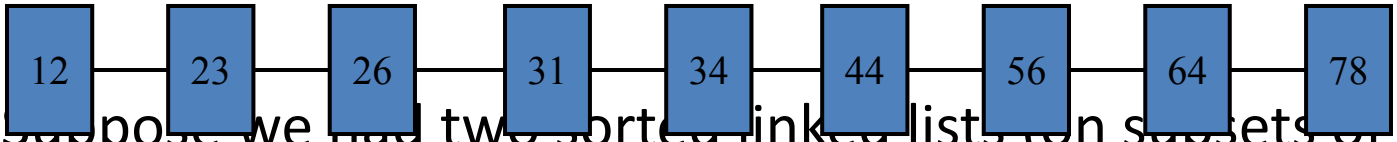
Skips Lists

(taken from http://u.cs.biu.ac.il/~amir/DS/w3_skip_lists_biu.ppt and https://users.cs.duke.edu/~reif/courses/alglectures/demleis.lectures/lec12.pdf)

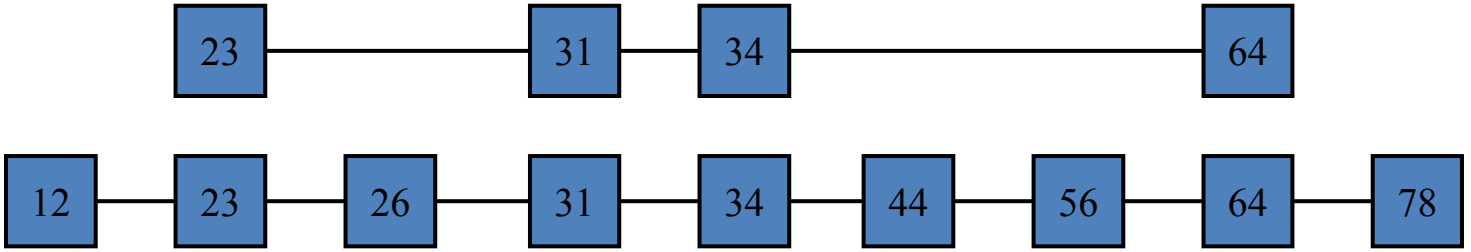
- A *skip list* is a data structure for dictionaries that uses a randomized insertion algorithm
  - Invented by William Pugh in 1989
- In a skip list with n entries
  - The expected space used is O(n)
  - The expected search, insertion and deletion time is O(log n)
- Skip lists are fast and simple to implement in practice

From list to skip list

- Start from simplest data structure: (sorted) linked list
- Search takes Θ(n) time in worst case
- How can we speed up searches?

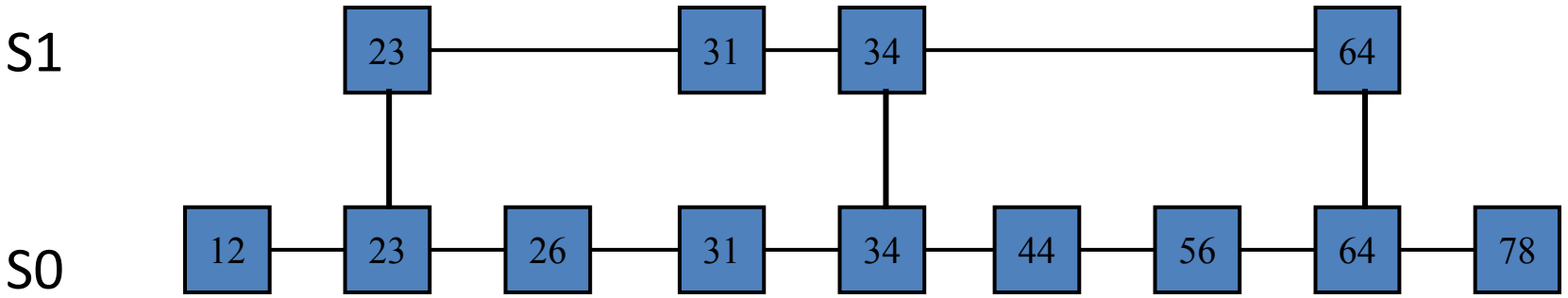


- Suppose we had two sorted linked lists (on subsets of the elements)
- Each element can appear in one or both lists • How can we speed up searches?



Linked lists as subway lines

- IDEA: Express and local subway lines (à la New York City 7th Avenue Line)
  - Express line connects a few of the stations
  - Local line connects all stations
  - Links between lines at common stations



Searching in two linked lists

SEARCH(x):

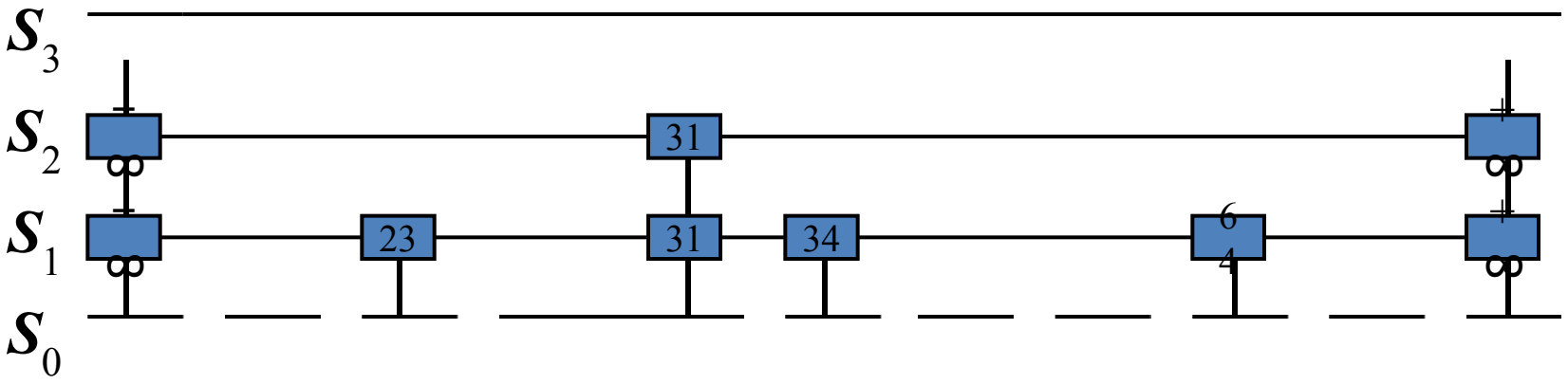
- Start with topmost list (S1) and walk right until until going right would go too far
- Walk down to lower linked list (S0) and walk right in S0 until element found (or not)

Choosing what nodes go to S1

- QUESTION: Which nodes should be in S1?
  - In a subway, the "popular stations"
  - Here we care about worst-case performance
    - Best approach: Evenly space the nodes in S1
    - But how many nodes should be in S1?
- ANALYSIS:
  - Search cost is roughly |S1| + |S1|/|S0|
  - Minimized (up to constant factors) when terms are equal: $|S1|^2 = |S0| = n \Rightarrow |S1| = \sqrt{n}$
  - Search cost is thus roughly $2\sqrt{n}$

More linked lists?

- What if we had more sorted linked lists?
- 2 sorted lists $\Rightarrow 2\sqrt{n}$
- 3 sorted lists $\Rightarrow 3\sqrt[3]{n}$
- k sorted lists $\Rightarrow \mathrm{k}\sqrt[k]{n}$
- lg n sorted lists $\Rightarrow \lg \mathrm{n}\ \sqrt[\lg n]{n} = 2\lg \mathrm{n}$
- lg n sorted linked lists are like a binary tree (in fact, level-linked B+-tree)

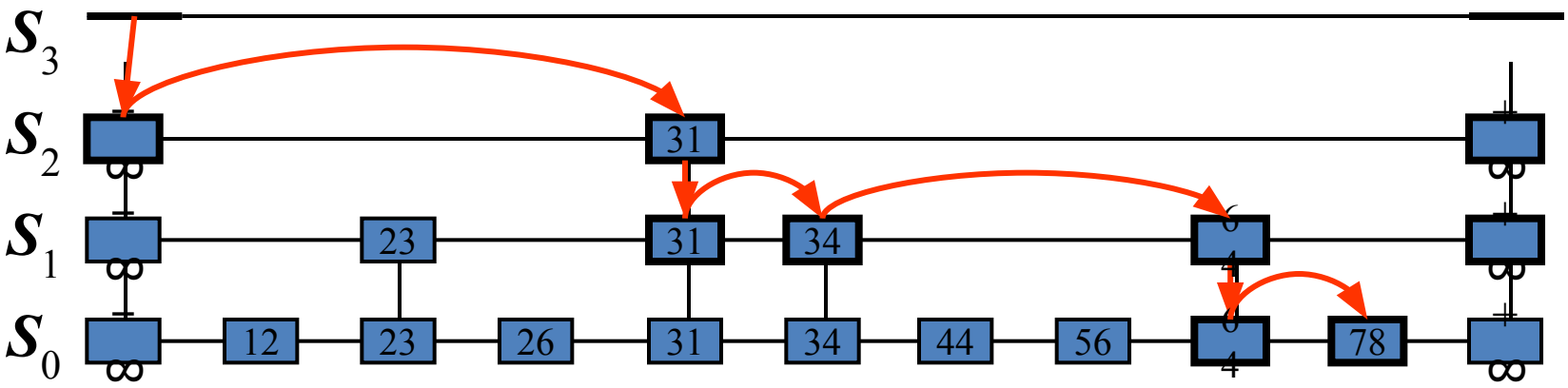


Skip Lists

- A skip list for a set S of distinct (key, element) items is a series of lists $S_0, S_1, \ldots, S_h$ such that
  - Each list S_i contains the special keys $+\infty$ and $-\infty$
  - List S_0 contains the keys of S in nondecreasing order
  - Each list is a subsequence of the previous one, i.e., $S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$
  - List S_h contains only the two special keys
- We show how to use a skip list to implement the dictionary ADT

Search

- We search for a key x in a a skip list as follows:
  - We start at the first position of the top list
  - At the current position p, we compare x with $y \leftarrow key(next(p))$
    - $x == y$: we return $element(next(p))$
    - $x > y$: we "scan forward"
    - $x < y$: we "drop down"
  - If we try to drop down past the bottom list, we return *null*
- Example: search for 78



© 2004 Goodrich Tamassia

Randomized Algorithms

- A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution
- It contains statements of the type

```
b ← random()
if b = 0
  do A …
else { b = 1}
  do B …
```

- Its running time depends on the outcomes of the coin tosses
- We analyze the expected running time of a randomized algorithm under the following assumptions
  - the coins are unbiased, and
  - the coin tosses are independent
- The worst-case running time of a randomized algorithm is often large but has very low probability (e.g., it occurs when all the coin tosses give "heads")
- We use a randomized algorithm to insert items into a skip list

© 2004 Goodrich Tamassia

Insert(x)

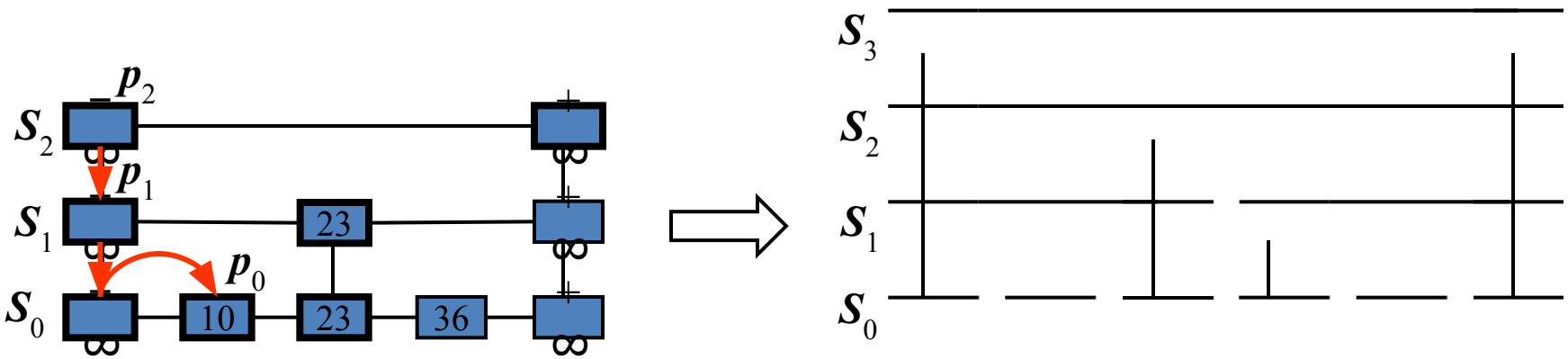
QUESTION: To which other lists should we add x?

IDEA:

- Flip a (fair) coin; if HEADS, promote x to next level up and flip again to see if x will be inserted in upper lists
- Let i be the number of consecutive HEADS
- When inserting x in a skip list, we search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with largest key less than x in each list $S_0, S_1, \ldots, S_i$
- For $j \leftarrow 0, \ldots, i$, we insert item (x, o) into list S_j after position p_j
- Probability of promotion to next level = 1/2
- On average:
  - 1/2 of the elements promoted 0 levels
  - 1/4 of the elements promoted 1 level
  - 1/8 of the elements promoted 2 levels
  - etc.

Example

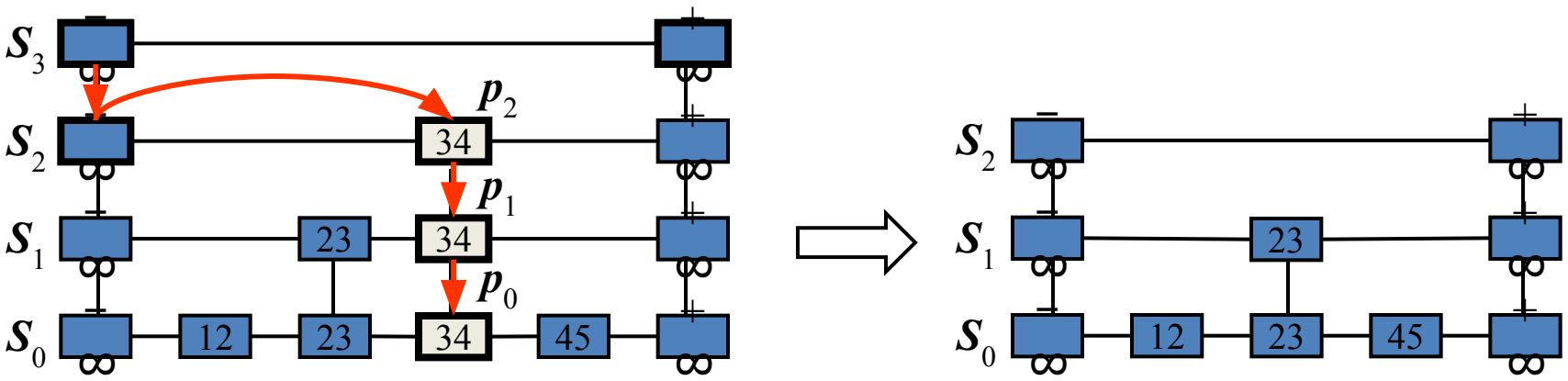
- Example: insert key 15, with HEADS, HEADS, HEADS, i.e. $i = 2$



© 2004 Goodrich Tamassia

Delete(x)

- To remove an entry with key x from a skip list, we proceed as follows:
  - We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with key x, where position p_j is in list S_j
  - We remove positions $p_0, p_1, \ldots, p_i$ from the lists $S_0, S_1, \ldots, S_i$
  - We remove all but one list containing only the two special keys
- Example: remove key 34



© 2004 Goodrich Tamassia

Summary

- A skip list is the result of insertions (and deletions) from an initially empty structure (containing just $+\infty$ and $-\infty$)
  - INSERT(x) uses random coin flips to decide promotion level
  - DELETE(x) removes x from all lists containing it
- THEOREM: With high probability, every search in an n-element skip list costs O(lg n)

Space Usage

- The space used by a skip list depends on the random bits used by each invocation of the insertion algorithm
- We use the following two basic probabilistic facts:

  Fact 1: The probability of getting i consecutive heads when flipping a coin is $1/2^i$

  Fact 2: If each of n entries is present in a set with probability p, the expected size of the set is np

- Consider a skip list with n entries
  - By Fact 1, we insert an entry in list S_i with probability $1/2^i$
  - By Fact 2, the expected size of list S_i is $n/2^i$
- The expected number of nodes used by the skip list is

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n$$

- Thus, the expected space usage of a skip list with n items is $O(n)$

© 2004 Goodrich Tamassia

Read more

- Read more at https://courses.csail.mit.edu/6.046/spring04/handouts/skiplists.pdf

Array ADT & Matrices

- An ***array*** is a sequence indexed by a system of integer coordinates
- The system of indexes is used to access elements and to permit alteration of individual elements.
- The elements can be accessed directly in O(1) time (constant time)
- The type array is built in any high level programming language, but we can still define our own type array and implement it in a different way
- Two integers `m` and `n`, `m:n` denotes all integers in the range `m..n`
- A matrix is a two-dimensional array but it can be extended to an arbitrary dimension
- In a `k`–dimensional array `A`, the index i_j in the dimension `j` ranges between $m_j : n_j$: $A[i_1, i_2, \ldots i_k]$ is the element of array `A` with index $(i_1, i_2, \ldots i_k)$

Matrix operation	Worst Case
Initialize a matrix with $r \times c$ elements with value x	O(r*c)
Return number of rows	O(1)
Return number of columns	O(1)
Lookup element at row i and column j	O(1)
Set a value for element at row i and column j	O(1)

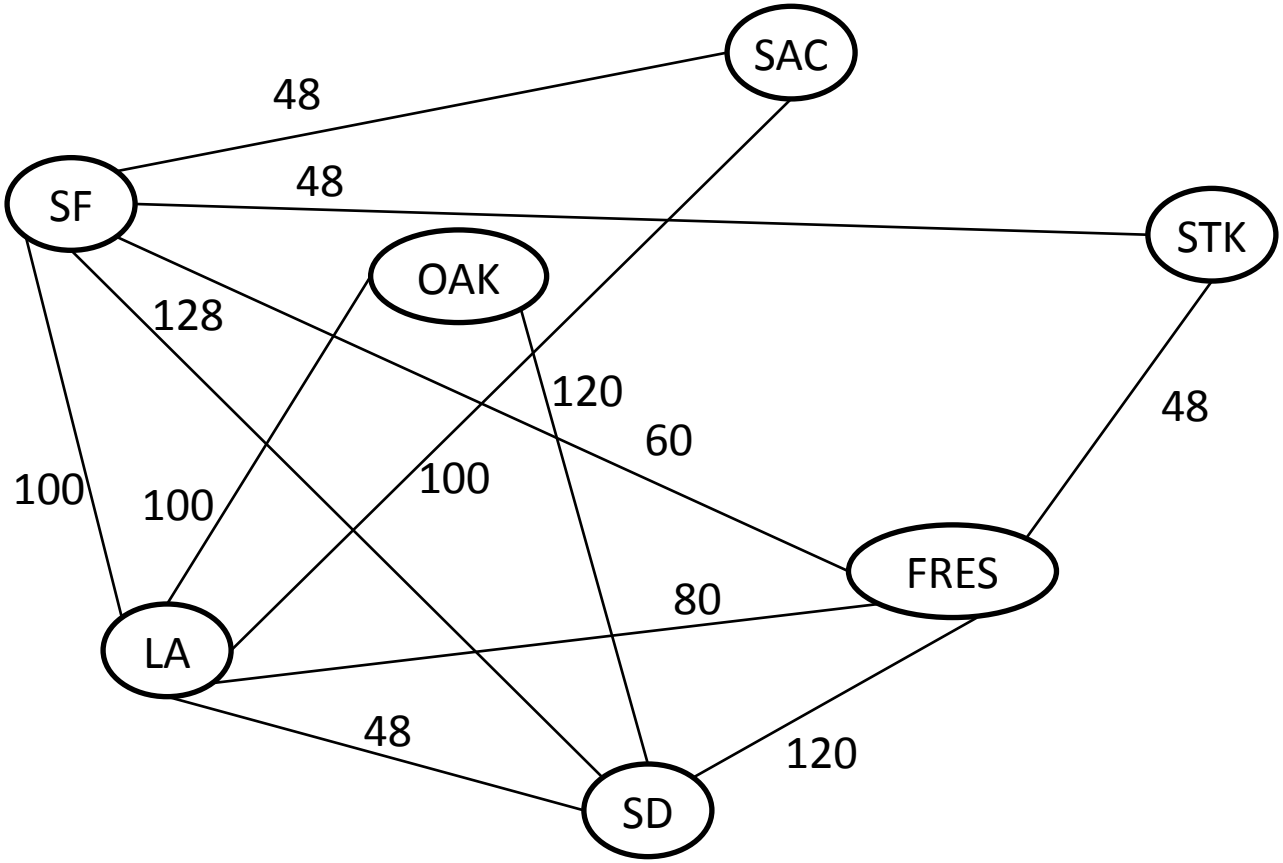
Graphs

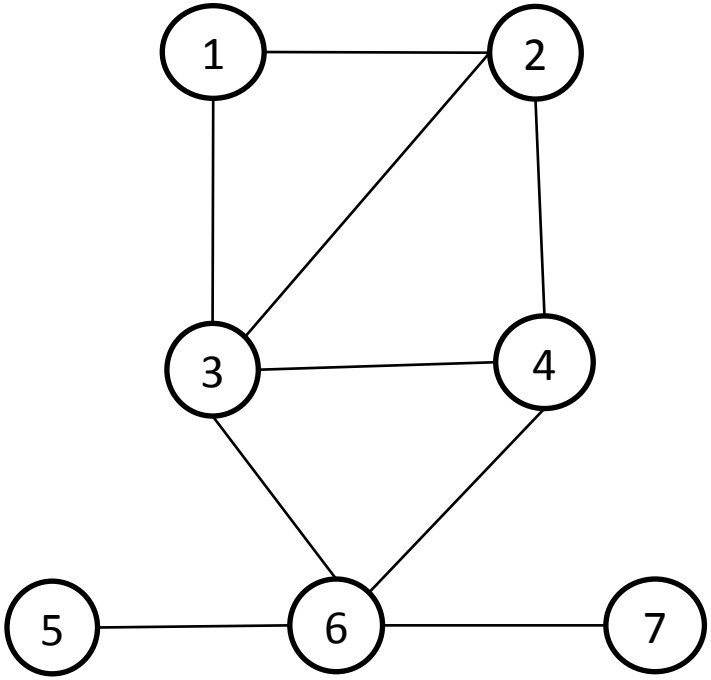
- Directed versus undirected versus weighted graphs
- Representation of a graph using an adjacency matrix or adjacency lists

- Undirected Graph
  - A *undirected graph* is a pair G = (V, E) where V is a set whose elements are called vertices, and E is a set of *unordered* pairs of *distinct* elements of V.
    - Vertices are often also called nodes.
    - Elements of E are called edges, or undirected edges.
    - Each edge may be considered as a subset of V containing two elements,
    - {v, w} denotes an undirected edge
    - In diagrams this edge is the line v---w.
    - In text we simply write vw, or wv
    - vw is said to be *incident* upon the vertices v and w

- Directed Graph
  - A *directed graph* (or *digraph*) is a pair G = (V, E) where V is a set whose elements are called vertices, and E is a set of ordered pairs of elements of V.
    - Vertices are often also called nodes.
    - Elements of E are called directed edges, or arcs.
    - For directed edge (v, w) in E, v is its tail and w its head;
    - (v, w) is represented in the diagrams as the arrow, v -> w.
    - In text we simply write vw.

- A weighted graph is a triple (V, E, W) where (V, E) is a graph (directed or undirected) and W is a function from E into R, the reals (integers or rationals).
  For an edge e, W(e) is called the *weight* of e.

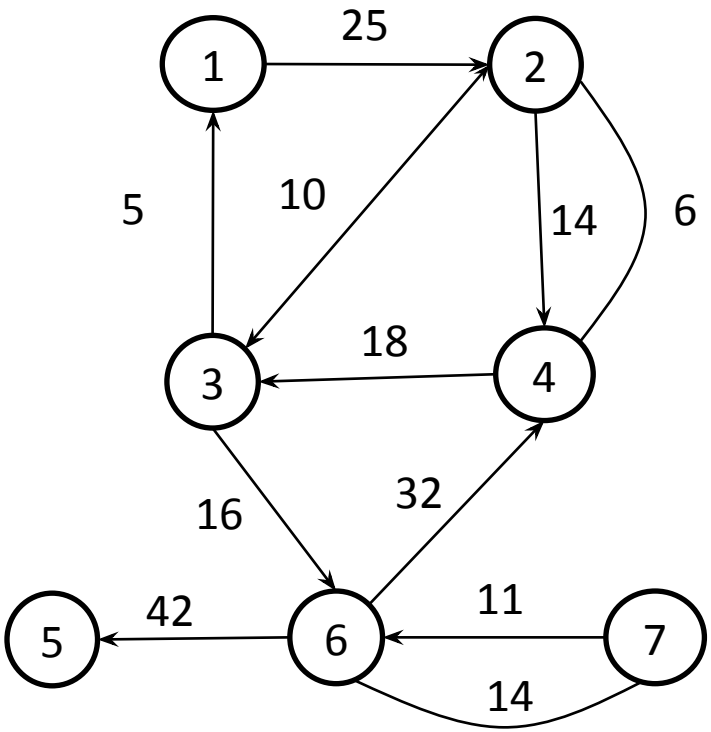




(a) An undirected graph

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

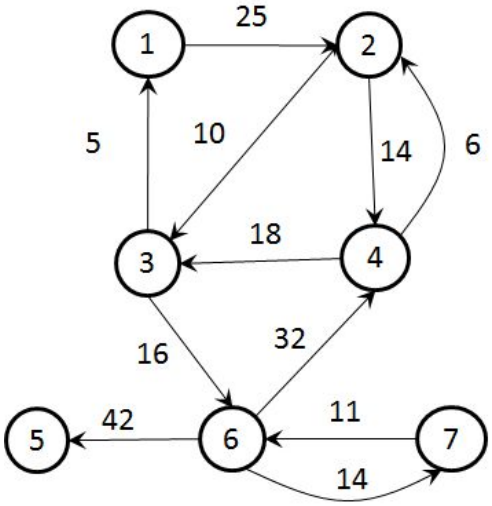
(b) Its adjacency matrix



(a) An weighted directed graph

$$\begin{bmatrix} 0 & 25 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 10 & 14 & \infty & \infty & \infty \\ 5 & \infty & 0 & \infty & \infty & 16 & \infty \\ \infty & 6 & 18 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 32 & 42 & 0 & 14 \\ \infty & \infty & \infty & \infty & \infty & 11 & 0 \end{bmatrix}$$

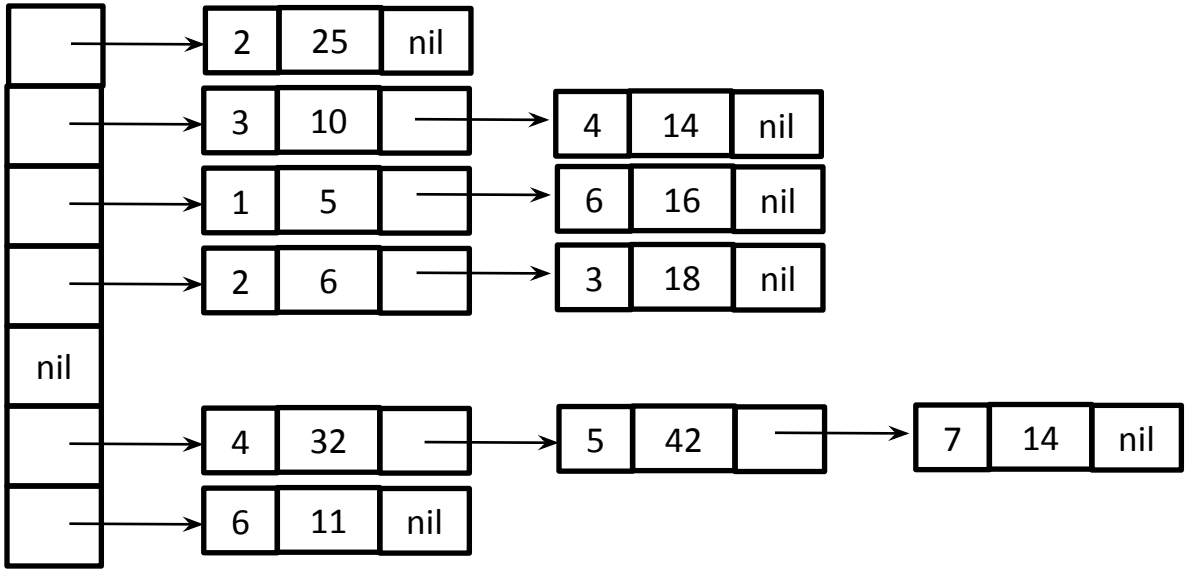
(b) Its weight matrix



(a) An weighted directed graph

$$\begin{bmatrix} 0 & 25 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 10 & 14 & \infty & \infty & \infty \\ 5 & \infty & 0 & \infty & \infty & 16 & \infty \\ \infty & 6 & 18 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 32 & 42 & 0 & 14 \\ \infty & \infty & \infty & \infty & \infty & 11 & 0 \end{bmatrix}$$

(b) Its weight matrix



(b) Its adjacency lists

State-space graph

(taken from Levitin, page 246-248)

- Many puzzles and games can be solved by a reduction to one of the standard graph problems.
  - vertices of a graph typically represent possible states of the problem in question, and
  - edges indicate permitted transitions among such states: one of the graph's vertices represents an initial state and another represents a goal state of the problem
  - The problem reduces to finding a path from the initial-state vertex to a goal-state vertex.
- Applications: state-space graphs are very important to AI
- Example: A peasant finds himself on a river bank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river in his boat. However, the boat has room only for the peasant himself and one other item (either the wolf, the goat, or the cabbage). In his absence, the wolf would eat the goat, and the goat would eat the cabbage. Find a way for the peasant to solve his problem or prove that it has no solution.

River crossing example

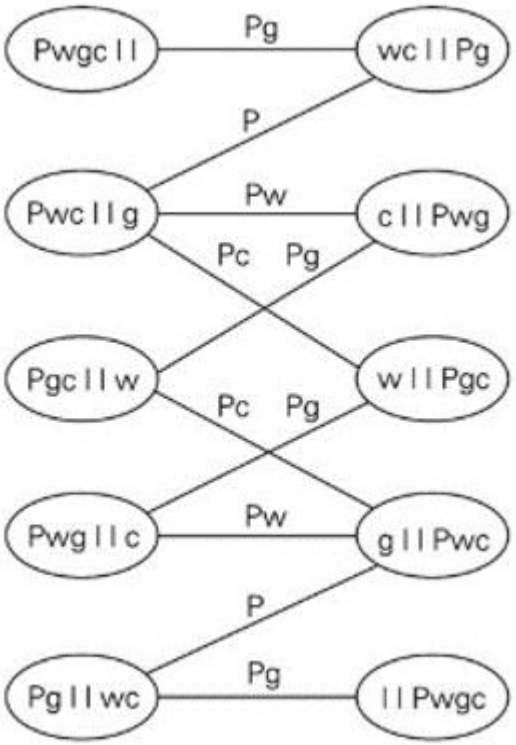


Fig. 6.18: P, w, g, c stand for the peasant, the wolf, the goat, and the cabbage, respectively; the two bars || denote the river; we label the edges by indicating the boat's occupants for each crossing.

- In terms of this graph, we are interested in finding a path from the initial-state vertex labeled Pwgc| | to the final-state vertex labeled | |Pwgc.
- Use BFS to find the shortest such paths; in this case, two distinct paths, each with 7 edges. i.e. river crossings

Trees

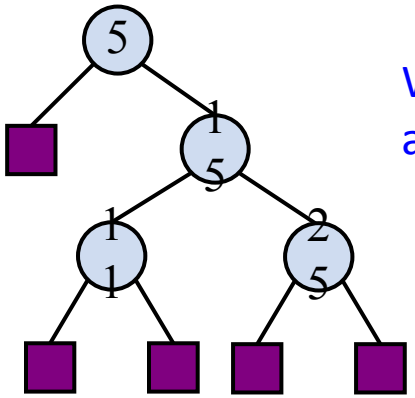
- A *tree* is a connected graph without cycles
- A *graph* is a pair of nodes and edges
- Examples
- Nodes are dots; an edge is a line connecting two nodes
- A *connected graph* is a graph in which between any two nodes there is a way of reaching one node from another
- Examples

- A cycle: there is a pair of nodes that has two disjoint ways of reaching one another
- Examples
- A tree that has a special node, called root, is called a rooted tree.
- In a rooted tree, each node except the root, has a parent
- Examples
- Each node may have child(ren)
- An external node (leaf) is a node without children
- An internal node is a node with at least one child; sometimes the root is not considered an internal node
- The *size* of a tree is the number of nodes in it

Depth

- Depth of a node: number of ancestors (between the node and the root)
  - Root has depth 0
- Depth of p's node is recursively defined:

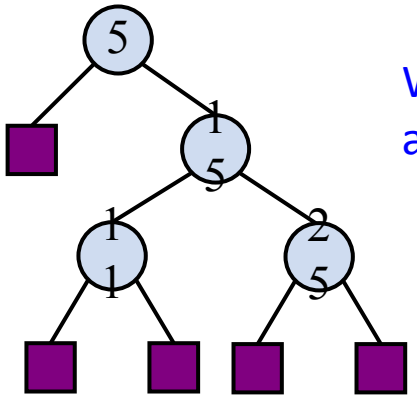
```
if (p.isRoot()) return 0;              // root has depth 0
else return 1 + depth(p.parent());   // 1 + (depth of parent)
```



What is the depth at each node?

Height

- Height of a node p in a tree T is defined recursively:
  - If p is the external node, then the height of p is 0
  - Otherwise, the height of p is 1 + the maximum height of children of p
- Height of a tree == maximum depth of its external nodes
  - Height of a tree T is the height of the root of T

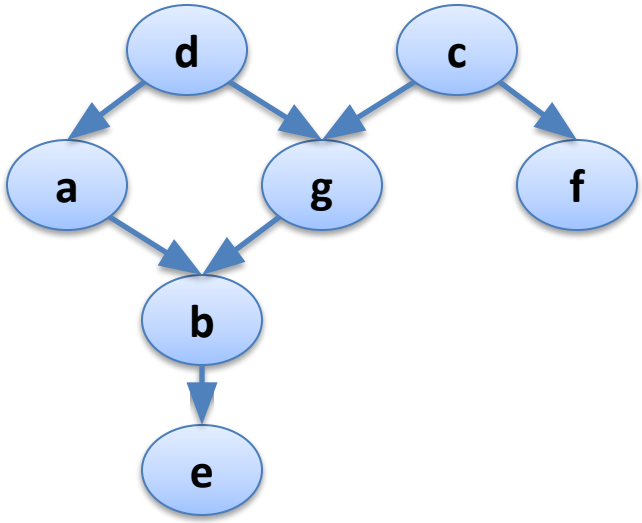


What is the height at each node?

Topological Sorting (application of DFS)

(slides taken from www.cs.toronto.edu/~tabrown/csc263/week9)

- A *Directed Acyclic Graph* (DAG) is a directed graph G = (V,E) without cycles.
- We use Depth First Search (DFS) traversal of a DAG starting at the first node in V
- We have a counter that is initialized to 0 and gets increments each time a node is visited
- For each node, we keep track of two timestamps, the discovery timestamp and the finishing timestamp
- Edges in a DFS traversal can be classified based on the discovery and finishing timestamps



DFS (Pages 603-610 of CLRS)

- We have a global counter called *time*=0 (initially)
- Each vertex v
  - has a color `v.color`: WHITE (initially), GREY (after discovered first time) or BLACK (all neighbors have been explored) and changes color WHITE→GREY→BLACK during DFS traversal
  - Has two timestamps: `v.d` records when v was discovered (and greyed) and `v.f` records when v was finished exploring (and blackened); always `v.d<v.f`

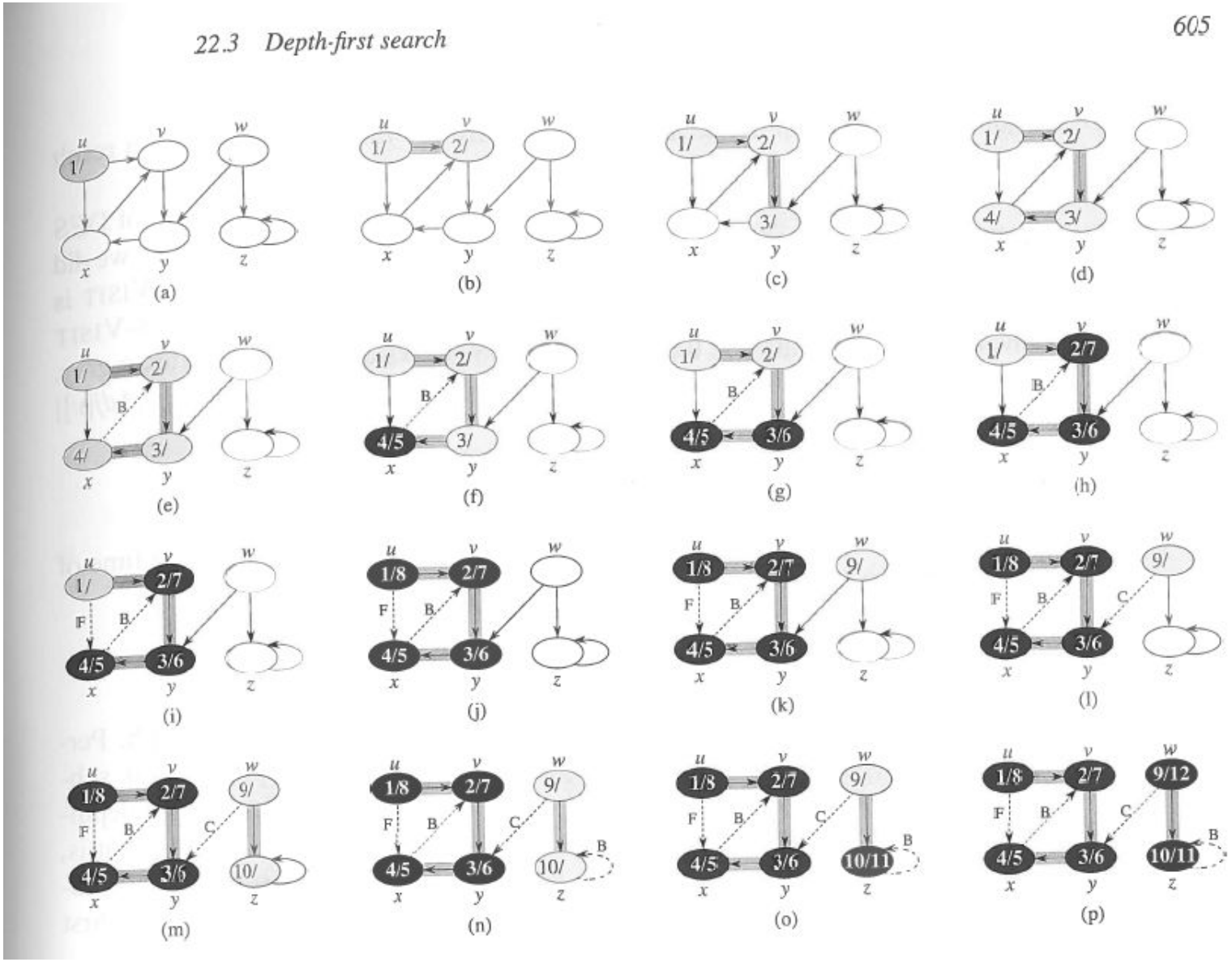
DFS Pseudocode (page 604)

```
DFS(G)
 for each vertex u ∈ G.V
     u.color = WHITE
     u.P = NIL
 time = 0
 for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(u)
```

```
DFS-VISIT(G,u)
 time = time + 1
 u.d = time
 u.color = GREY
 for each vertex v ∈ G.Adj[u]
     if v.color == WHITE
         v.P = u
         DFS-VISIT(G,v)
 u.Color = BLACK
 time = time + 1
 u.f = time
```

Example (pg 605 CLRS)

tree edges are shaded, the rest are dotted

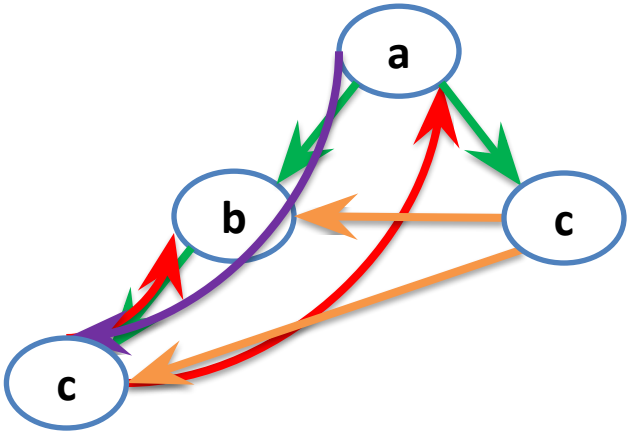


Edge classification by DFS

- The edges in E will be partitioned into 4 types:
- Edge (u,v) is a:
  - *Tree edge* iff u discovers v during the DFS so u becomes parent of v, v.P = u
  - *Forward edge* iff u is an ancestor of v in the DFS tree
  - *Back edge* iff u is a descendant of v in the DFS tree
  - *Cross edge* iff u is neither an ancestor nor a descendant of v

Edge classification by DFS

Tree edges
Forward edges
Back edges
Cross edges

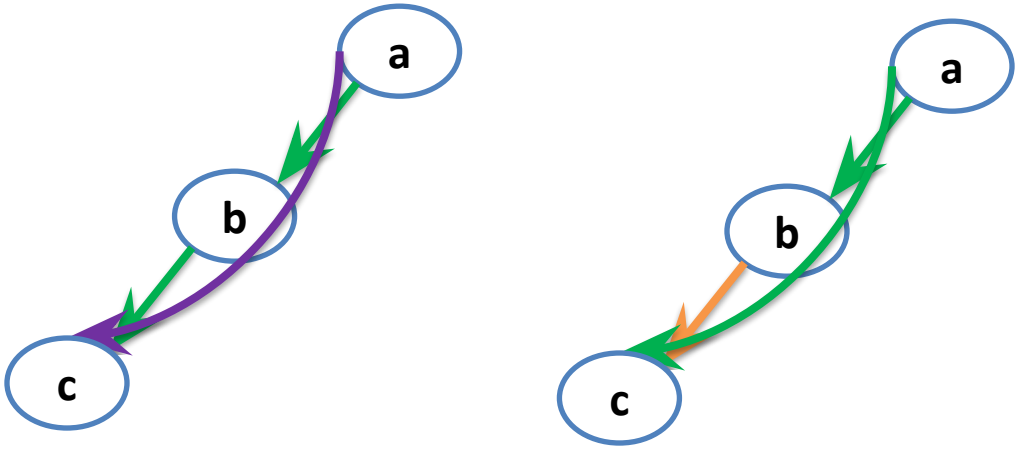


The edge classification depends on the particular DFS tree!

Edge classification by DFS

Tree edges
Forward edges
Back edges
Cross edges

Both are valid



The edge classification depends on the particular DFS tree!

DAGs and back edges

- Can there be a **back** edge in a DFS on a DAG?
- NO! Back edges close a cycle!
- A graph **G** is a DAG <=> there is no back edge classified by DFS(**G**)

Topological Sort

- A *topological sort* of a DAG G=(V,E) is a linear ordering of all its vertices (aka a permutation of V) such that if G contains an edge (u,v), then u appears before v in its ordering.
- Typically topological sort renames each vertex with IDs from 1..n
- Applications:
  - We have a set of tasks and a set of dependencies (precedence constraints) of form "task A must be done before task B". We need an ordering of the tasks that conforms with the given dependencies such that each edge $(u, v) \in E$ means that task u must be done before task v.
  - When scheduling task graphs in distributed systems, usually we first need to sort the tasks topologically and then assign them to resources (the most efficient scheduling is an NP-complete problem).
  - During compilation to order modules/libraries. For example, command `apt-get` uses topological sorting to obtain the admissible sequence in which a set of Debian packages can be installed/removed.

Algorithm for TS (page 613 of CLRS)

- TOPOLOGICAL-SORT(**G**):
  1) call DFS(G) to compute **finishing** times **f**[**v**] for each vertex **v**
  2) as each vertex is finished, insert it onto the **front** of a linked list
  3) return the linked list of vertices

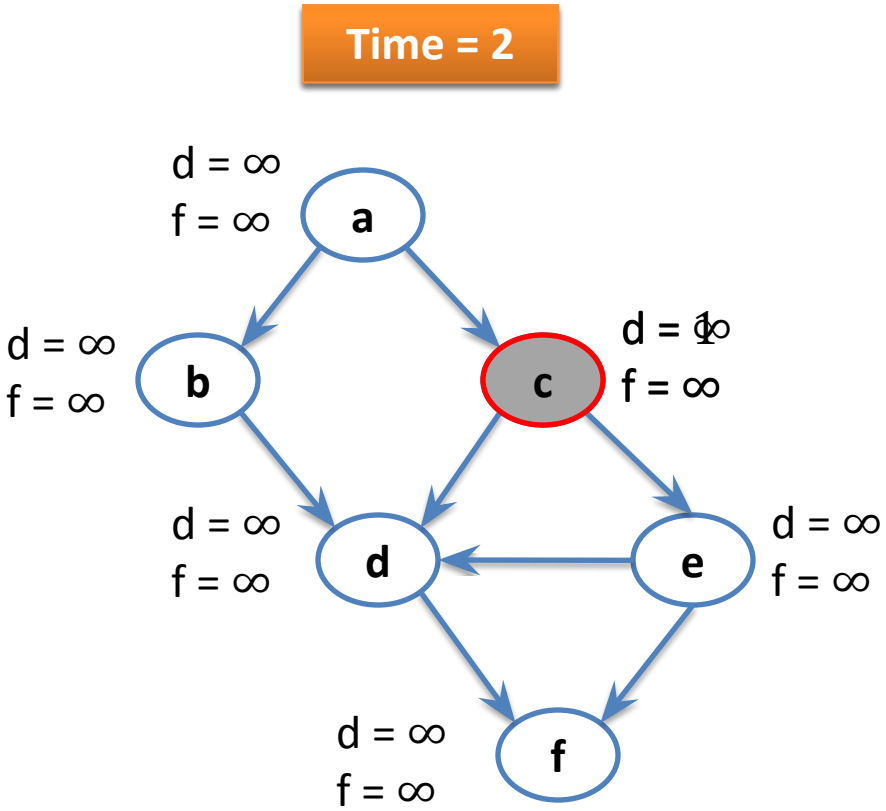
- Note that the result is just a list of vertices in order of **decreasing** finish times **f**[]

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

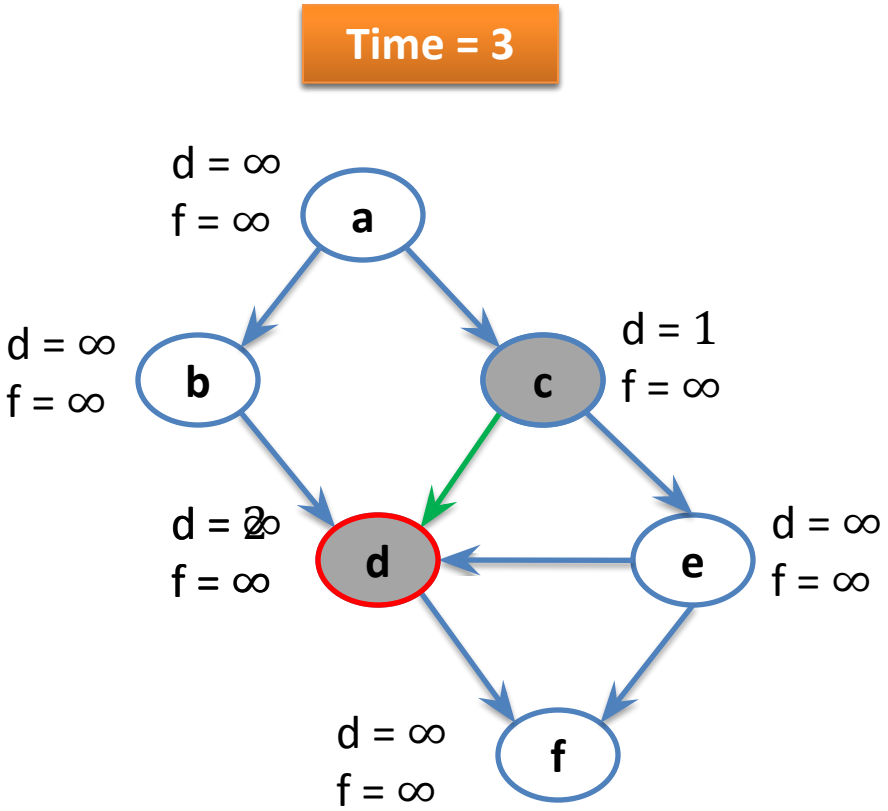


Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Let's say we start the DFS from the vertex **c**

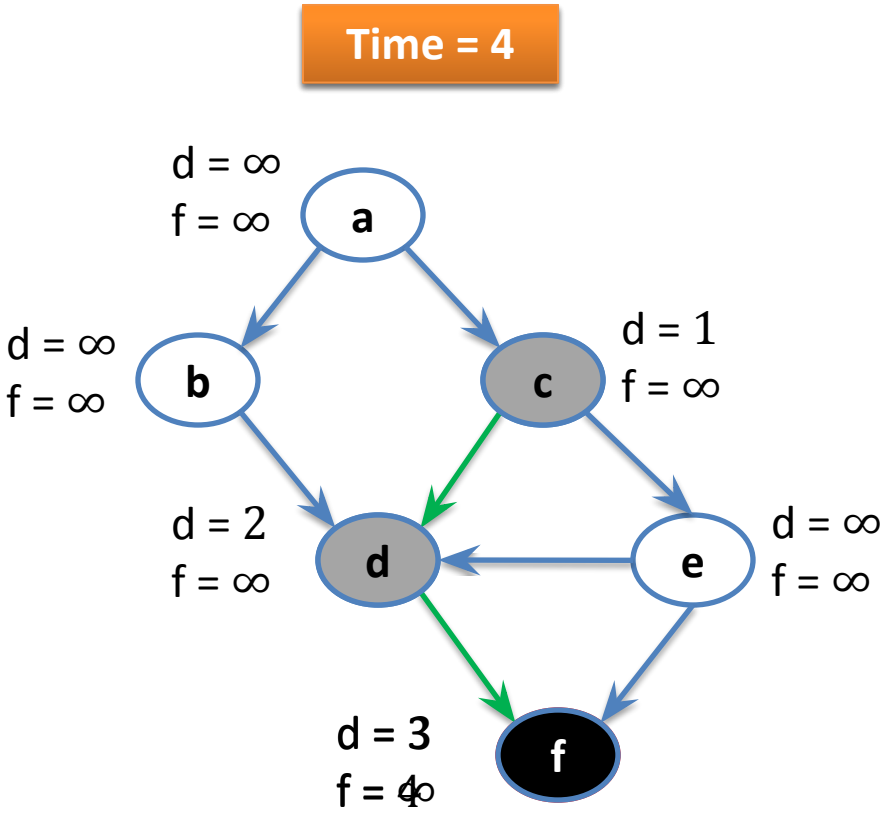
Next we discover the vertex **d**



Topological sort

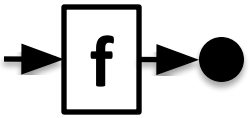
1) Call DFS(**G**) to compute the finishing times **f**[**v**]

2) as each vertex is finished, insert it onto the **front** of a linked list



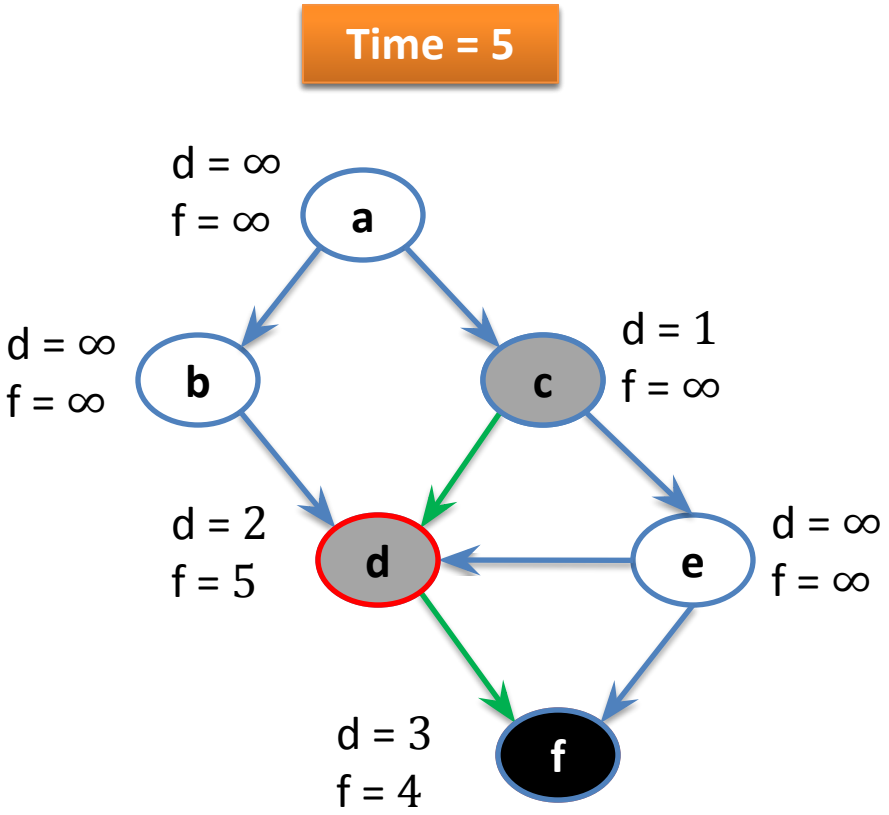
Next we discover the vertex **f**

f is done, move back to **d**



Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



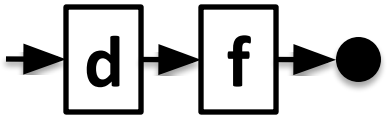
Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Next we discover the vertex **f**

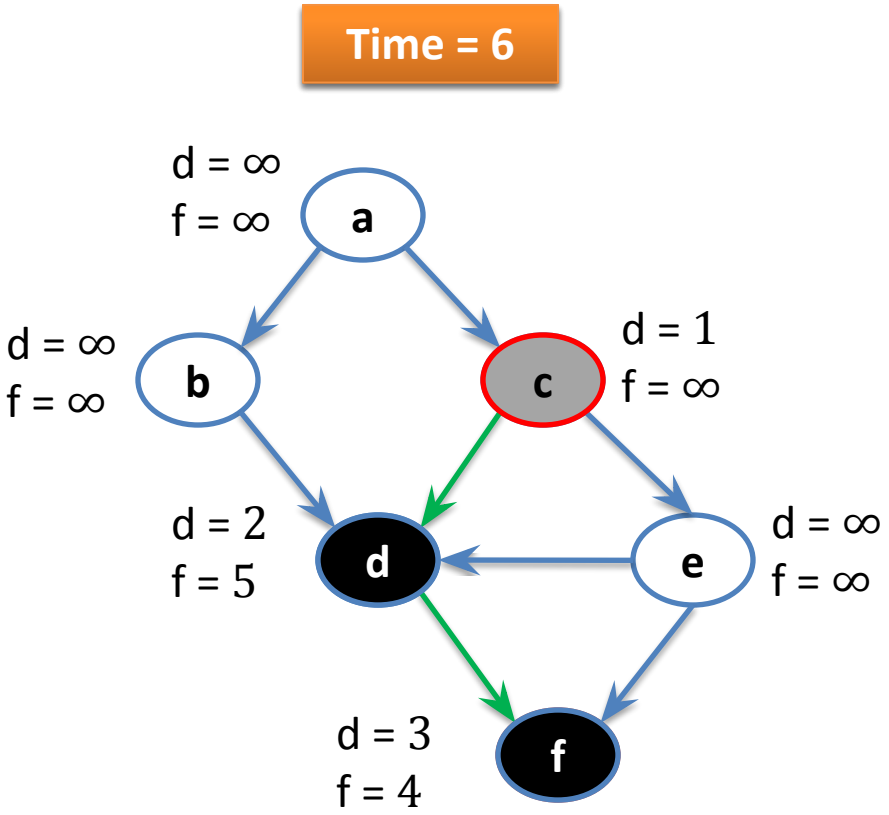
f is done, move back to **d**

d is done, move back to **c**



Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



Let's say we start the DFS from the vertex **c**

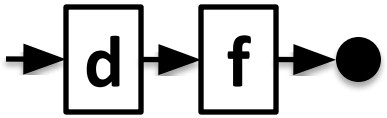
Next we discover the vertex **d**

Next we discover the vertex **f**

f is done, move back to **d**

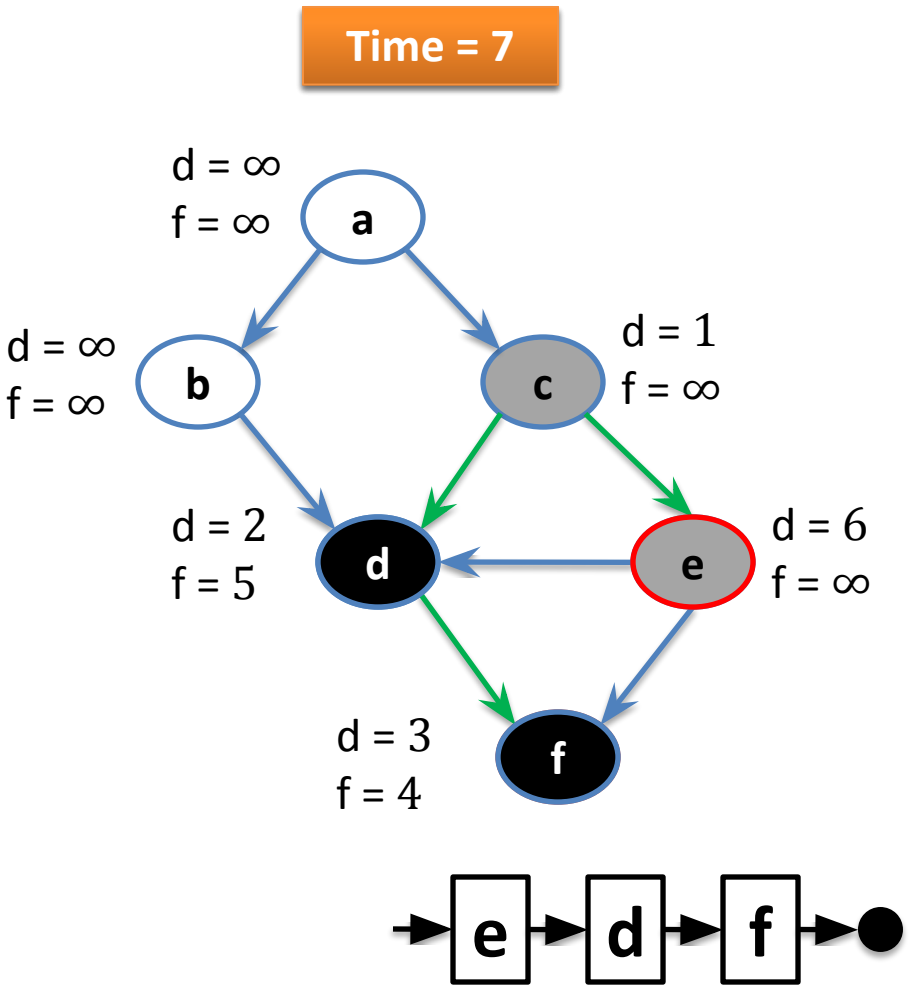
d is done, move back to **c**

Next we discover the vertex **e**



Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Both edges from **e** are **cross edges**

d is done, move back to **c**

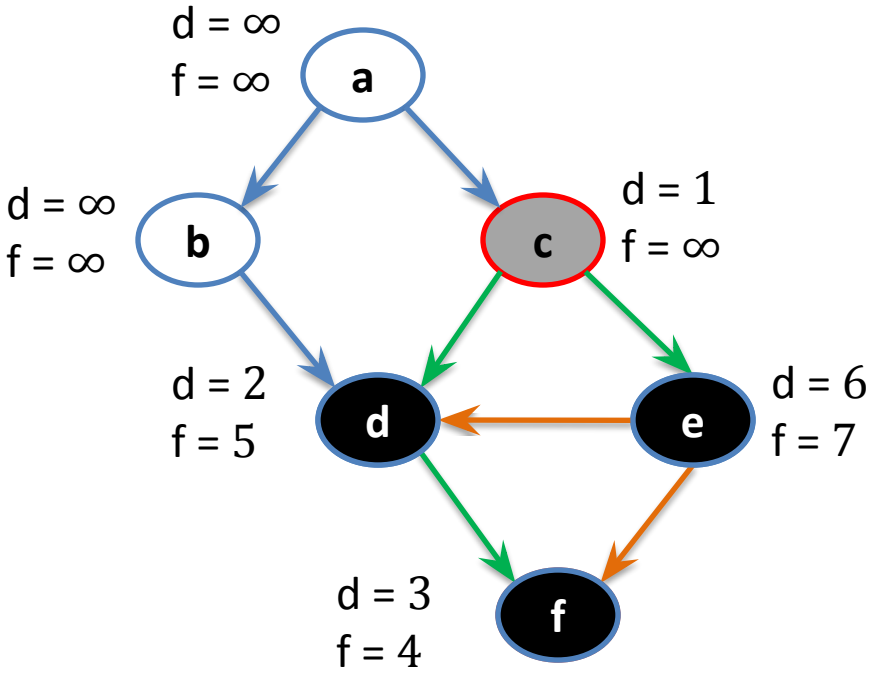
Next we discover the vertex **e**

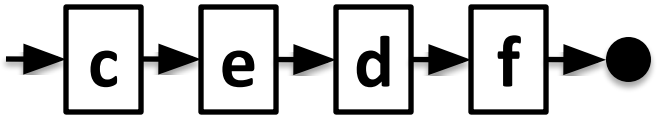
e is done, move back to **c**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 8





Let's say we start the DFS from the vertex **c**

Just a note: If there was (**c**,**f**) edge in the graph, it would be classified as a **forward edge** (in this particular DFS run)

d is done, move back to **c**

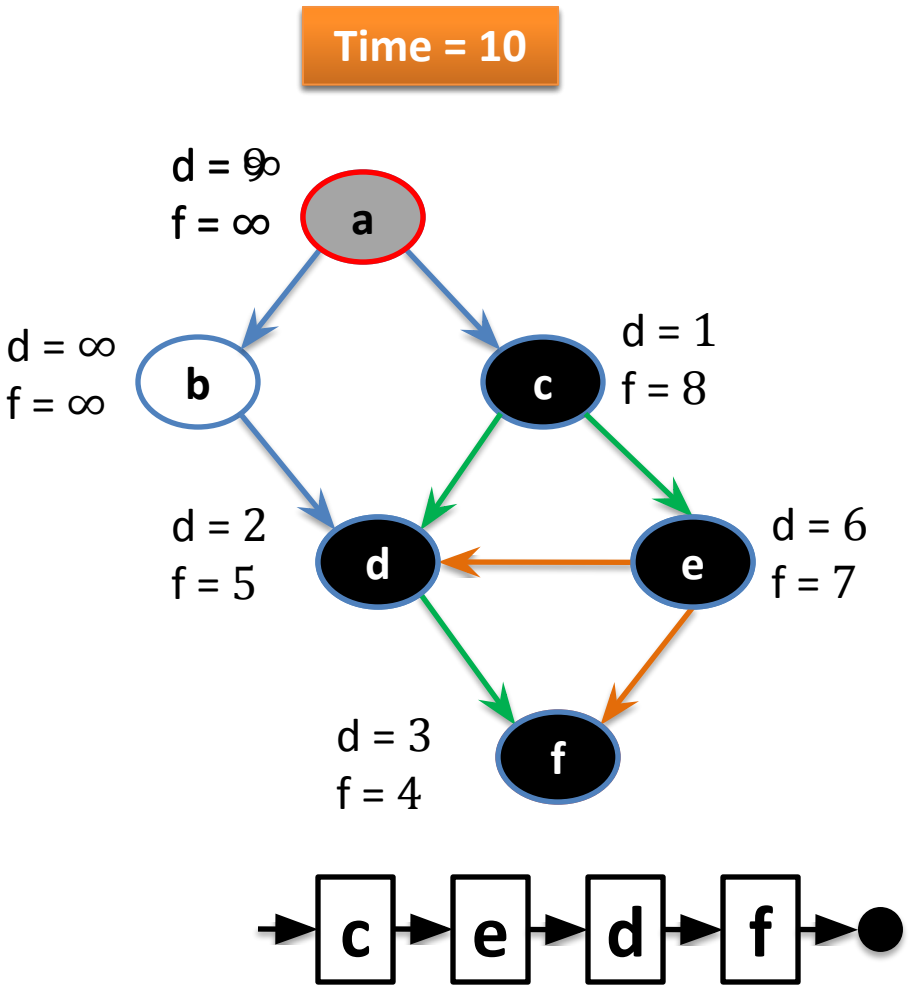
Next we discover the vertex **e**

e is done, move back to **c**

c is done as well

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



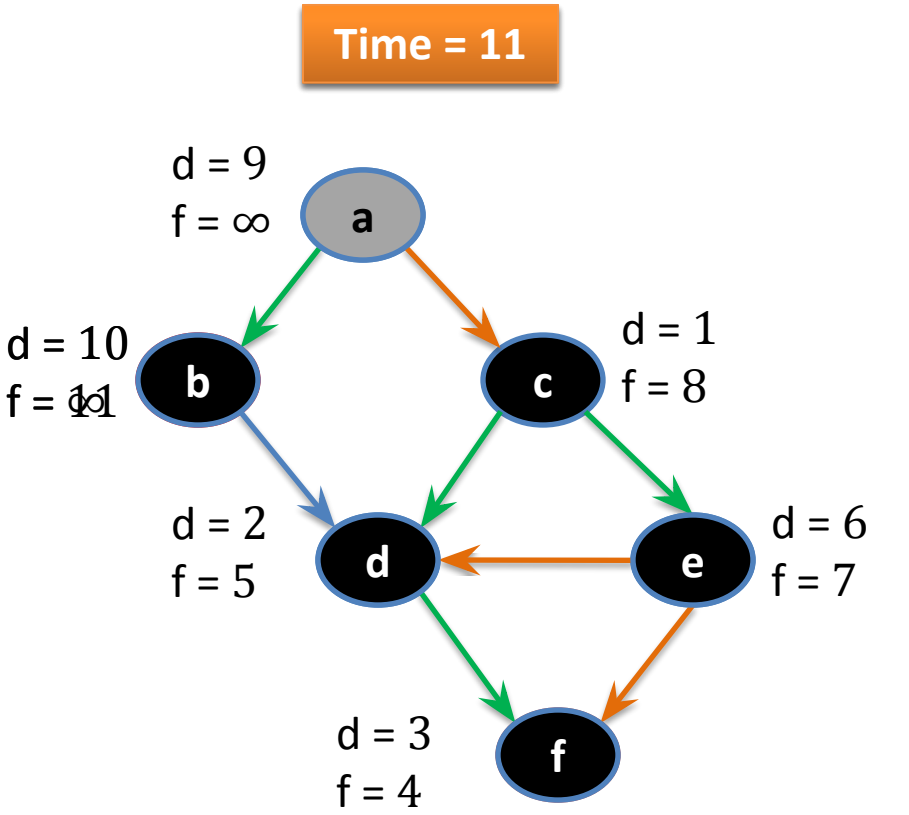
Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

Next we discover the vertex **b**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

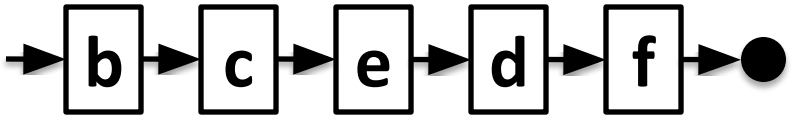


Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

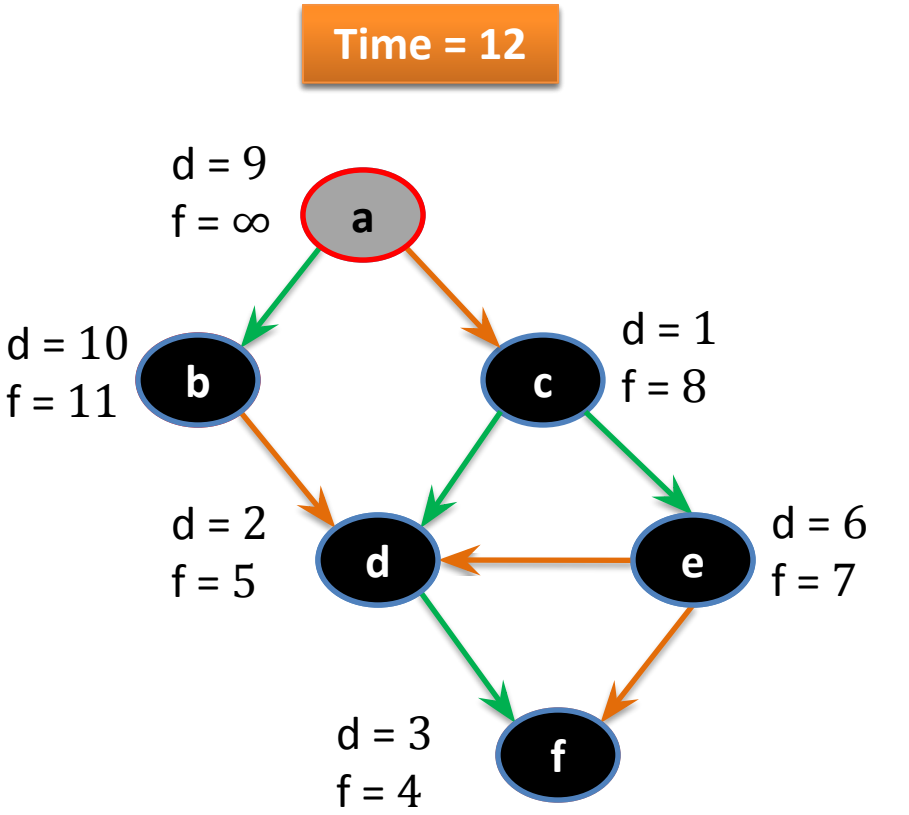
Next we discover the vertex **b**

b is done as (**b**,**d**) is a cross edge => now move back to **c**



Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



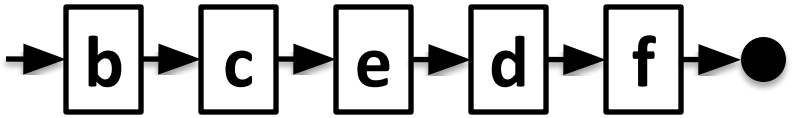
Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

Next we discover the vertex **b**

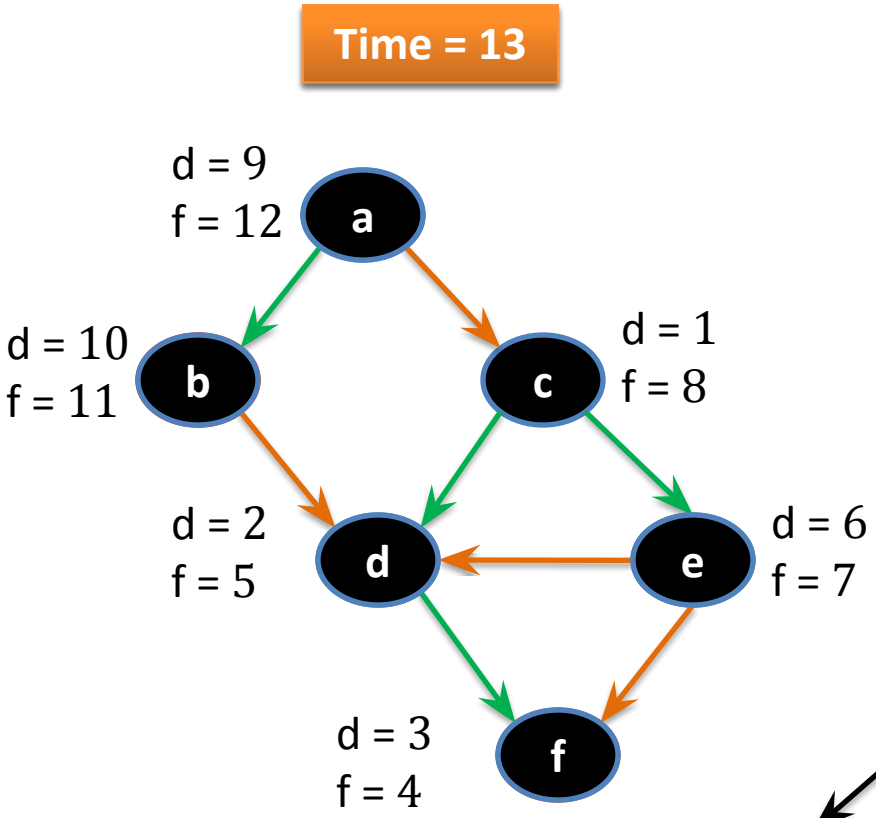
b is done as (**b**,**d**) is a cross edge => now move back to **c**

a is done as well



Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



WE HAVE THE RESULT!

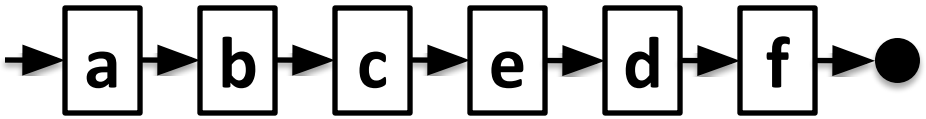
3) return the linked list of vertices

=> (**a**,**c**) is a cross edge

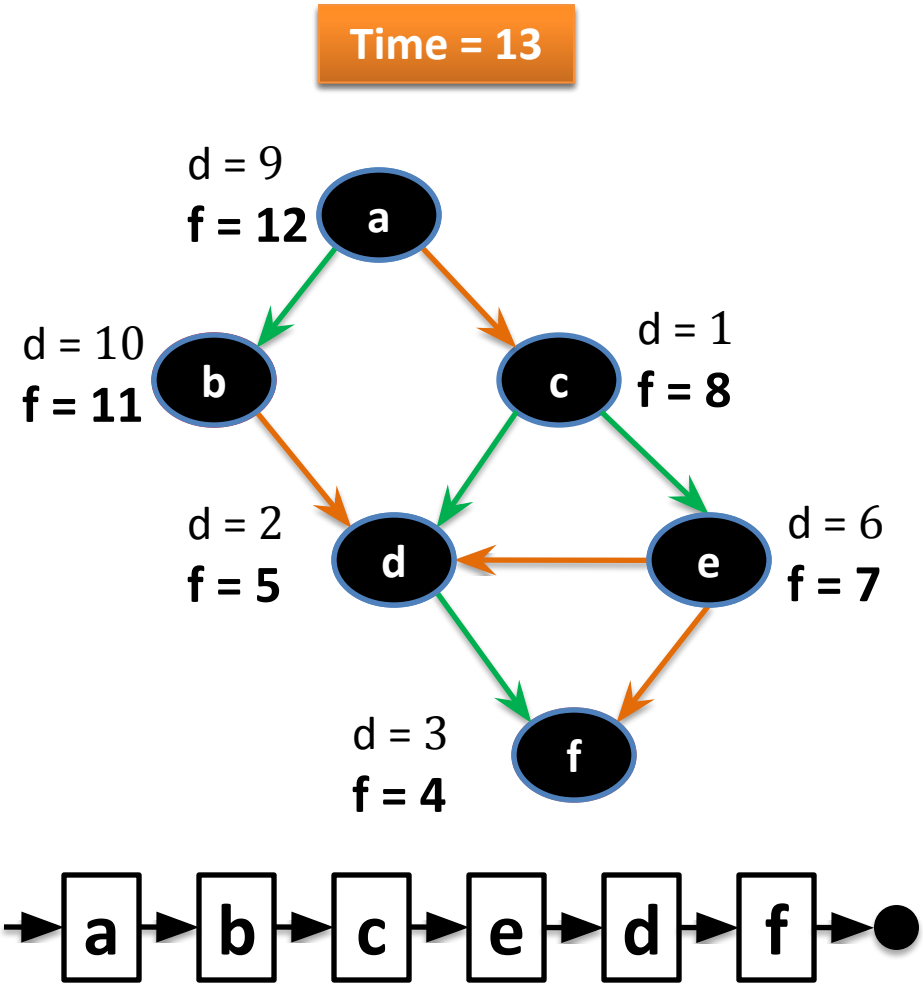
Next we discover the vertex **b**

b is done as (**b**,**d**) is a cross edge => now move back to **c**

a is done as well



Topological sort



The linked list is sorted in **decreasing** order of finishing times **f**[]

Try yourself with different vertex order for DFS visit

Note: If you redraw the graph so that all vertices are in a line ordered by a valid topological sort, then all edges point „**from left to right**“

Time complexity of TS(G)

- Running time of topological sort:

$$\Theta(n + m)$$

where $n = |V|$ and $m = |E|$

- Why? Depth first search takes $\Theta(n + m)$ time in the worst case, and inserting into the front of a linked list takes $\Theta(1)$ time

Proof of correctness

- **Theorem**: TOPOLOGICAL-SORT(**G**) produces a topological sort of a DAG **G**

- The TOPOLOGICAL-SORT(**G**) algorithm does a DFS on the DAG **G**, and it lists the nodes of **G** in order of decreasing finish times **f**[]
- We must show that this list satisfies the topological sort property, namely, that for every edge (**u**,**v**) of **G**, **u** appears before **v** in the list
- **Claim**: For every edge (**u**,**v**) of **G**: **f**[**v**] < **f**[**u**] in DFS

Proof of correctness

"For every edge (**u**,**v**) of **G**, **f**[**v**] < **f**[**u**] in this DFS"

- The DFS classifies (**u**,**v**) as a **tree edge**, a **forward edge** or a **cross-edge** (it cannot be a back-edge since **G** has no cycles):
  i. If (**u**,**v**) is a **tree** or a **forward edge** $\Rightarrow$ **v** is a descendant of **u** $\Rightarrow$ **f**[**v**] < **f**[**u**]
  ii. If (**u**,**v**) is a **cross-edge**

Proof of correctness

"For every edge (**u**,**v**) of **G**: **f**[**v**] < **f**[**u**] in this DFS"

ii. If (**u**,**v**) is a **cross-edge**:

Q.E.D. of Claim

- as (**u**,**v**) is a cross-edge, by definition, neither **u** is a descendant of **v** nor **v** is a descendant of **u:**

d[**u**] < f[**u**] < d[**v**] < f[**v**]

or

d[**v**] < f[**v**] < d[**u**] < f[**u**]

since (**u**,**v**) is an edge, **v** is surely discovered **before** **u**'s exploration completes

f[v] < f[u]

Proof of correctness

- TOPOLOGICAL-SORT(G) lists the nodes of G from highest to lowest finishing times

- By the **Claim**, for every edge (**u**,**v**) of **G**:

  f[**v**] < **f**[**u**]

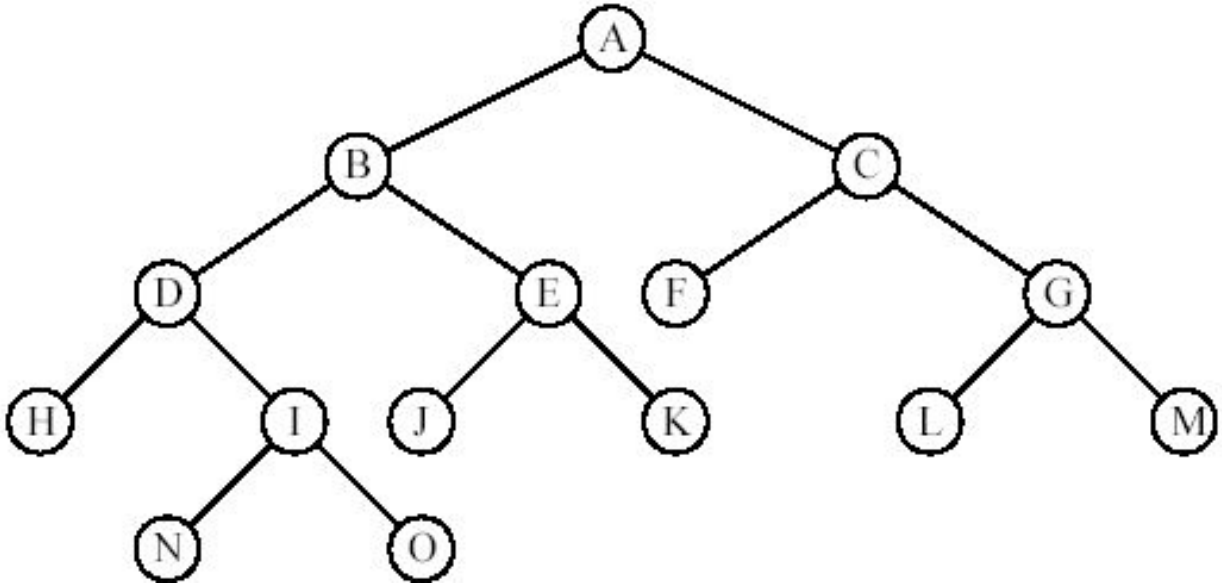
⇒ **u** will be before **v** in the algorithm's list

- Q.E.D of **Theorem**

Binary Trees

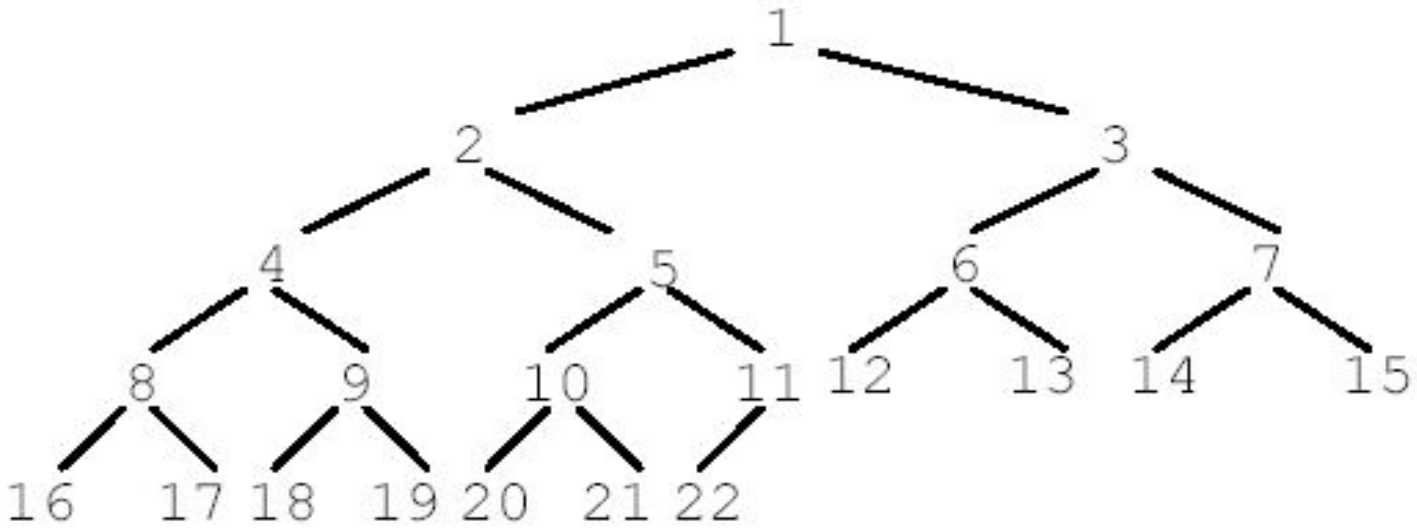
- A binary tree is a rooted, oriented tree in which each node has at most two children
  - A child is a left child or a right child
  - In the tree's traversal, left children precede right children
- There are left subtrees and right subtrees
- In a full binary tree (also called proper binary tree or 2-tree), every node has either 0 or 2 children
- In a complete tree, every level is complete except maybe the last level where nodes are missing from right to left

full binary tree (extended binary tree) is an ordered tree consisting of two types of nodes: external nodes with no children and internal nodes with exactly two children.



- ***Property 3*** A full binary tree with N internal nodes has N+1 external nodes.

Complete binary tree is a full binary tree where the external nodes on the bottom level appear from the left to right.



Property 4 The height of a complete binary tree with N internal nodes is about $log_2 N$

Binary Tree Traversal

- Is a procedure that systematically visits every vertex of an ordered rooted tree
- Tree traversals are naturally recursive
- Since a binary tree has three "parts," there are six possible ways to traverse the binary tree:

root, left, right left, root, right

left, right, root root, right, left

right, root, left right, left, root

- Three most used algorithms
  - Preorder traversal.
  - Inorder traversal.
  - Postorder traversal.

Preorder traversal

- In preorder, the root is visited *first*
- The code for preorder traversal to print out all the elements in the binary tree:

```
void preorderPrint(const Tree& T) {
  cout << T.element; // print element
        if (T.leftChild != NULL)
              preorderPrint (T.leftChild);
        if (T.rightChild != NULL)
              preorderPrint(T.rightChild);
  }
```

Inorder traversal

- In preorder, the left child is visited *first*, then the root, then the right child
- The code for inorder traversal to print out all the elements in the binary tree:

```
void inorderPrint(const Tree& T) {
 if (T.leftChild != NULL)
        inorderPrint (T.leftChild);
    cout << T.element; // print element
    if (T.rightChild != NULL)
        inorderPrint(T.rightChild);
 }
```

Postorder traversal

- In preorder, the root is visited *last*, after the left child, then the right child
- The code for postorder traversal to print out all the elements in the binary tree:

```
void postorderPrint(const Tree& T) {
 if (T.leftChild != NULL)
        postorderPrint (T.leftChild);
    if (T.rightChild != NULL)
        postorderPrint(T.rightChild);
    cout << T.element; // print element
 }
```

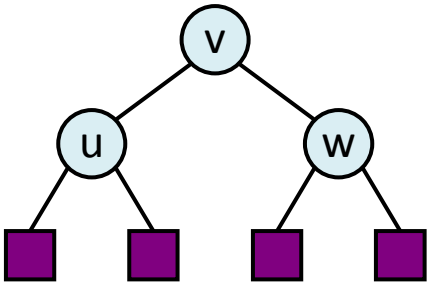
Binary Search Tree

- BST = a tree in which each node is an object
- In addition a node contains fields `left`, `right` and `p` that point to the left child, right child and parent, respectively; if a node is missing, then the field contains NIL
- The root is the only node for which p is NIL
- The keys satisfy the *binary-search-tree property*:

Let `x` be a node in the BST. If `y` is a node in the left subtree of `x` then `key[y]` < `key[x]`. If `y` is a node in the right subtree of `x` then `key[x]` < `key[y]`.

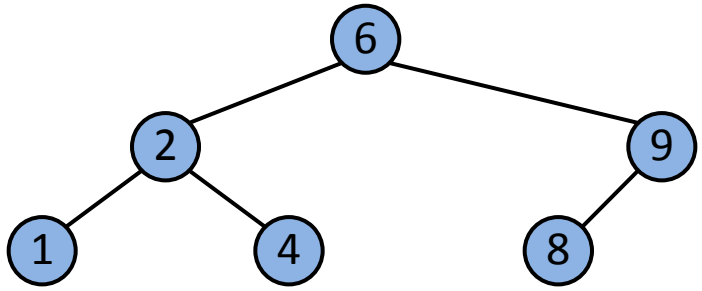
- Query operations supported by binary search trees: Search(x, k), Minimum(x), Maximum(x), Successor(x), Predecessor(x) where `x` is a node and `k` is a key value
- They run in `O(h)` time where `h` is the height of the tree

Binary Search Trees (BST)



- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let ***u***, ***v***, and ***w*** be three nodes such that ***u*** is in the left subtree of ***v*** and ***w*** is in the right subtree of ***v***. We have
    key(***u***) ≤ ***key***(***v***) ≤ ***key***(***w***)

- An inorder traversal of a binary search trees visits the keys in increasing order

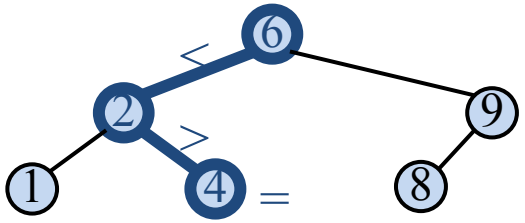


Operations

- Search: locate a node with a specific key
- Insert: add a node
- Remove: remove a node

Search in a BST

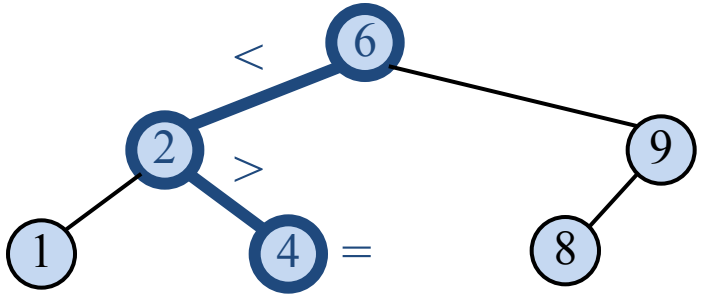
- To find/search for a key k, we trace a downward path starting at the root
- Navigate left and right down the tree
- At each node, compare node key to search key
- If they match, search terminates successfully
- If search key is less, go down to left subtree
- If search key is greater, go down to right subtree
- If no match has been found when bottom of tree is reached (no left or right subtree to descend to), search terminates unsuccessfully.



BST Search

- Example: search(4):

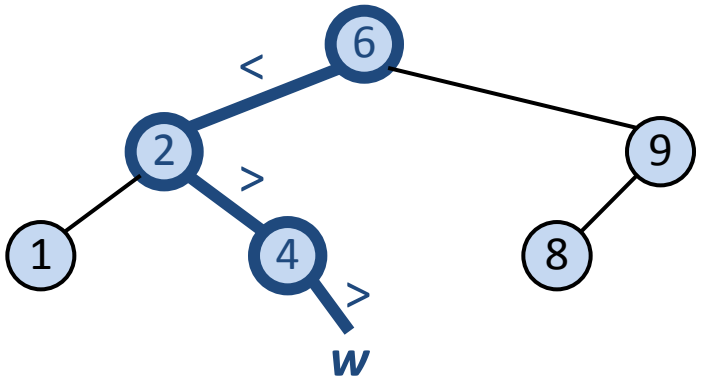
Call BSTSearch(4)

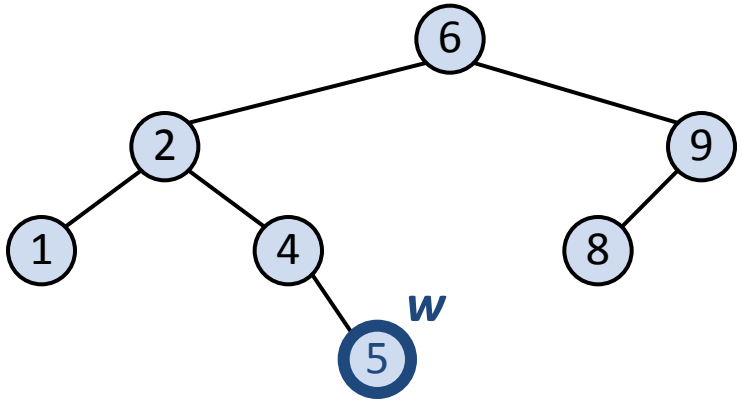



```
BSTSearch(key) {
  cur = root
  while (cur is not null)
    if (key == cur->key)
      return cur // Found
    else if (key < cur->key)
      cur = cur->left
    else
      cur = cur->right
  return null // Not found
}
```

Insert in a BST

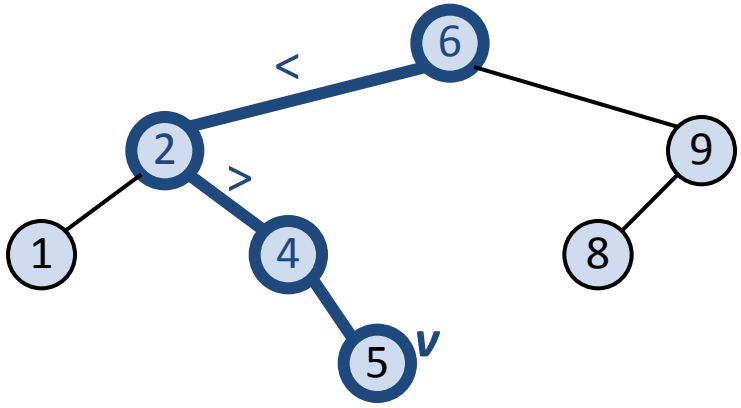
- To perform operation insert(k, val), we search for key k
- Assume k is not already in the tree, and let w be the null pointer reached by the search
- Create a new node with key k and attach it to pointer w
- Example: BSTInsert (5)





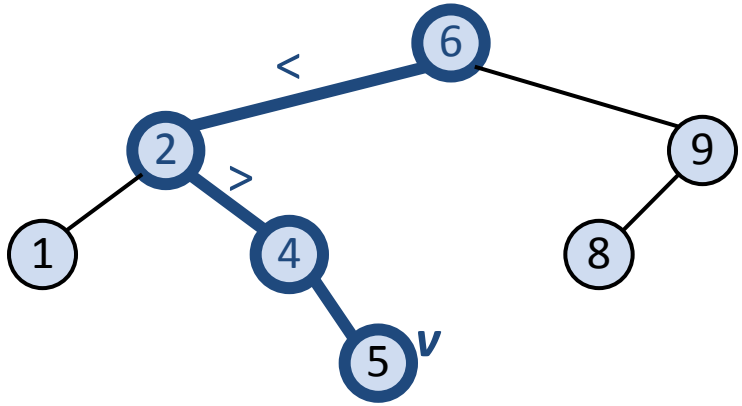
Delete from a BST

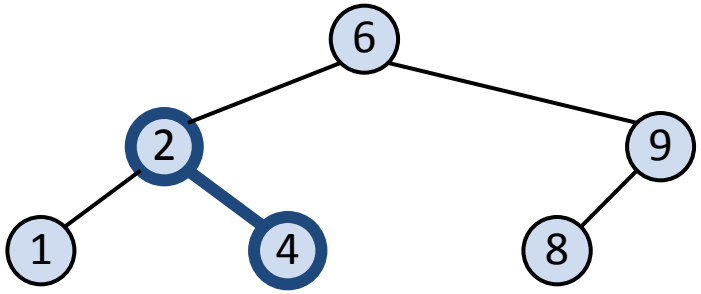
- Most difficult operation
- If the node to be removed is not a leaf, one should keep its child(ren)
- Three cases for the node to be deleted: the node has
  - no children (aka a leaf)
  - 1 child
  - 2 children



BST Remove – 0 children

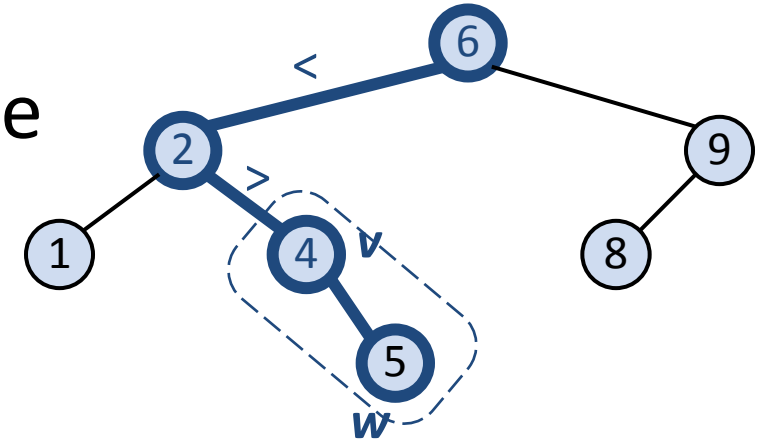
- To perform operation erase(k), we search for key k
- Assume key k is in the tree, and let let v be the node storing k
- If node v has NO internal children, just remove v
- Example: remove 5

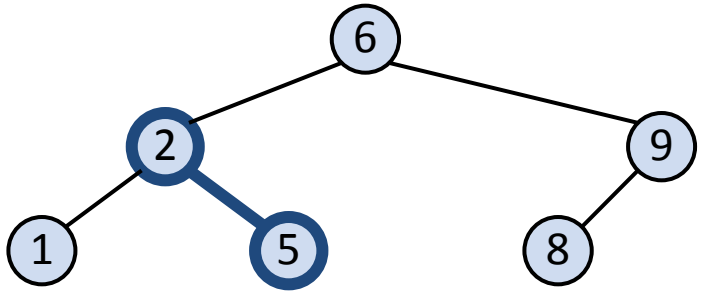




BST Remove – 1 internal child

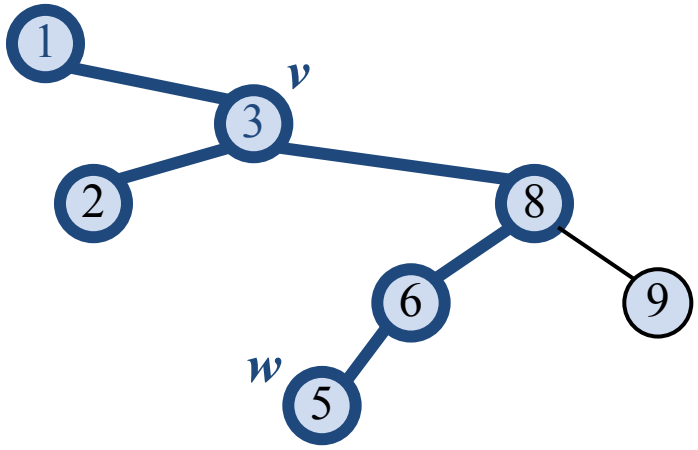
- If node v has one child w, we remove v and attach w as the child of v's parent
- "Ask grandparent to take care of single child"
- Example: remove (4)

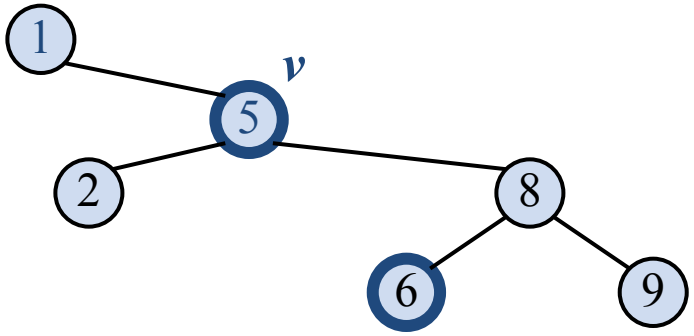




BST Remove – 2 internal children

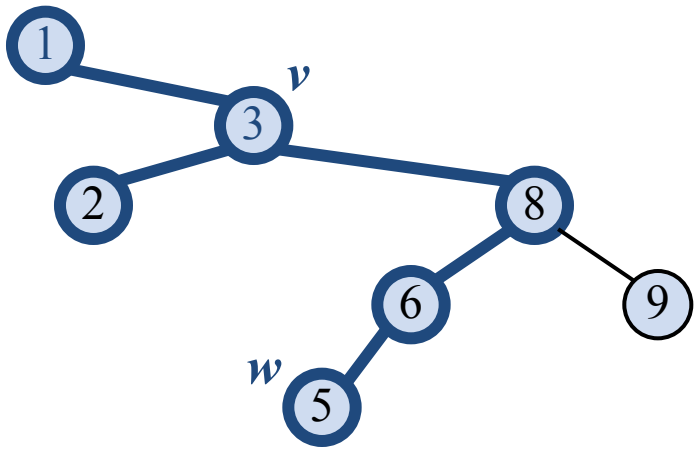
- Consider the case where the key ***k*** to be removed is stored at a node ***v*** whose children are both internal
  - Find the internal node ***w*** that follows ***v*** in an inorder traversal
  - Copy ***key***(***w***) into node ***v***
  - Remove node ***w*** – this node will have at most one child
- Example: remove (3)





BST Remove – 2 internal children (contd.)

- How to find the internal node ***w*** that follows ***v*** in an inorder traversal?
  - Find ***v***'s successor by going right, then left, left,…
- Examples:
- Successor of 3 is 5
- Successor of 8 is 9



Number of BST trees

- The total number of binary search trees with n keys is equal to the nth Catalan number, $c(n) = \frac{1}{n+1}\binom{2n}{n}$ for n>0, c(0)=1.

which grows to infinity as fast as $4^n/n^{1.5}$

Dynamic Programming

- Applicable to only a narrow range of problems, but offers impressive efficiency gains
- The pattern is suited to problems with a divide-and-conquer structure (e.g. decrease-by-one or decrease-by-half) where there is “overlap” between the sub-instances
- Instead of re-computing each sub-instance, keep the result in an array called the *cache* and access it when needed
- An array (list) T is used as a cache for sub-instance solutions
- The first time a particular sub-instance is solved, its solution is stored in T
- Later, when needed, directly access the solution by a fast array lookup rather than re-compute it from scratch

- A dynamic programming algorithm is analyzed by its *dimension* = the number of nested for loops needed to *fully initialize* the cache T **and** *solve* the full-sized problem

- The 1D (one-dimensional) dynamic programming pattern

```
def dynamic_programming_1D(instance):
 T = one dimensional empty array
 <INITIALIZE BASE CASE ELEMENTS OF T>
 for i =<LEAST NON-BASE CASE> to <GREATEST INSTANCE> do
    <INITIALIZE T[i] USING ONLY ELEMENTS OF T THAT HAVE
     ALREADY BEEN INITIALIZED>
 return T[<SOLUTION INDEX>]
```

All-pairs Shortest Paths

- *Shortest path problems* involve computing minimal-weight paths between vertices in weighted graphs.
- In a weighted graph, each edge may have a numeric weight: < 0, zero or > 0.
- Numeric weights may be unconstrained or be required to be non-negative.
- *All-pairs shortest path problems* involve computing paths between all pairs of distinct vertices.

All-pairs Shortest Paths

- Problem:

input: a weighted graph $G = (V, E)$ with no negative cycles

output: a two-dimensional array *distance*

$$distance[u, v] = \begin{cases} d_{u,v} & \text{if u and } v \text{ are connected, or} \\ \infty & \text{otherwise} \end{cases}$$

- Simplistic approach: run a single-pair shortest path algorithm on all relevant pairs of vertices
  - Extending Dijkstra's algorithm: if no negative weight edges, then the algorithm has $O(|V|^3 \log V)$ time complexity and uses priority queues
  - Extending Bellman-Ford-Moore algorithm: if there are no negative weight cycles, then the algorithm has $O(|V|^2 \cdot |E|)$ time complexity

Floyd-Warshall Algorithm

- Also known as Floyd's algorithm, the Roy-Warshall algorithm, the Roy-Floyd algorithm, or the WFI algorithm.
- Finds the shortest paths between all pairs of nodes in a weighted graph with no negative weight cycles
- It does not return details of the paths themselves
- It is similar to Bellman-Ford-Moore algorithm: incrementally improves an estimate on the shortest path between two vertices until the estimate is optimal.

Transitive closure of a graph

- The transitive closure of a directed graph with n vertices can be defined as the n × n boolean matrix T = {t_{ij} }, in which the element in the i th row and the jth column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the jth vertex; otherwise, t_{ij} is 0.
- An example of a digraph, its adjacency matrix, and its transitive closure is given in Figure 8.11.

- *Optimal Substructure Property*: given a shortest path, any subpath is also a shortest path between corresponding nodes
- Let `V={1,2,…,N}` be the set of nodes in the graph `G`
- Let function `shortestPath(i,j,k)` return the shortest path from `i` to `j` using vertices only from the set `{1,2,...,k}` as intermediate points along the way
- Our goal: shortest path in `G` between `i` and `j` is the return of function `shortestPath(i,j,N)`
- Dynamic programming: calculate `shortestPath(i,j,k+1)` using the previous calculated shortest paths; it could be either

  (1) a path that only uses vertices in the set {1, ..., k} => `shortestPath(i,j,k)`

  (2) a path that goes from i to k + 1 and then from k + 1 to j => `shortestPath(i,k+1,k)+shortestPath(k+1,j,k)`

- Compute shortestPath(i, j, k) for all (i, j) pairs starting with k = 1, then k = 2, … until k=N.
- Floyd-Warshall algorithm:

```
for i = 1 to |V| do
  for j = 1 to |V| do
     dist[ i, j ] = ∞
for i = 1 to |V| do
  dist[ i, i ] = 0
for each edge (u,v) do
  dist[ u, v ] = w(u,v)  // the weight of the edge (u,v)
for k = 1 to |V| do
  for i = 1 to |V| do
   for j = 1 to |V| do
      if dist[ i, j ] > dist[ i, k ] + dist[ k, j ] then
         dist[ i, j ] = dist[ i, k ] + dist[ k, j ]
```

- Take the example from Wikipedia:

https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

- If there are negative cycles in the graph, the Floyd–Warshall algorithm can be used to detect them
- The path between a node and itself `dist[i,i]` should remain 0
- Initially, `dist[i,i]=0`
- F-W algorithm tried to improve any path, including the one between a node and itself
- The value of `dist[i,i]=0` will be negative if there exists a negative-length path from `i` back to `i`
- One can inspect the diagonal of the matrix `dist`, and the presence of a negative number indicates that the graph contains at least one negative cycle

Longest path in a graph

- The longest path problem is NP-hard and the decision version of the problem, which asks whether a path exists of at least some given length, is NP-complete.
- However, the longest path problem has a linear time solution for directed acyclic graphs.
- The idea is similar to the linear time solution for shortest path in a directed acyclic graph by using Topological Sorting.

Pseudocode

Longest-path-DAG(G)
Input: Unweighted DAG G = (V, E)
Output: Longest path in G
1: Topologically sort G
2: Initialize array dist[] = {MIN_INT, ….MIN_INT} // size N
3: dist[0] = 0
4: for each vertex u $\in$ V in linearized order do
// compute dist(u) = $\max_{(v,u)\in E}$ {dist(v) + 1}
5: for every v $\in$ V do
6: if (A[u][v] == 1 && dist[v] < dist[u] + 1)
7: then dist[v] = dist[u] + 1
8: endif
9: enddo
10: endfor
11: return $\max_{v\in V}$ {dist(v)}

Maximum subarray problem

- Proposed by Ulf Grenander in 1977

input: an array $\langle p_1, p_2, \cdots, p_n \rangle$ where each $p_i \in \mathbb{R}$ is a profit t (or loss) on day *i*

output: indices s, e with s ≤ e, maximizing the total profit

$$\sum_{i=s}^{e} p_i$$

Applications:

- buy then sell a stock/security
- pick opening/closing time of a retail store with slow periods
- computer vision, data mining: identify region most consistent with a pattern e.g. street striping

Greedy fails

Straightforward greedy algorithm would be:

- buy at the lowest price or sell at the highest price
- incorrect; best "run" could be elsewhere
- example: -3, -2, 4, -1, -2, 1, 5, 4, -3
- Greedy suggests: -3, **-2, 4, -1, -2, 1, 5**, 4, -3
- But optimal is -3, -2, **4, -1, -2, 1, 5, 4**, -3

Exhaustive optimization, aka brute force

- Always give the optimal solution, but high time complexity
- try every legal start/end

```
1: function BRUTE-FORCE-MAX-SUBARRAY(P)
2:    s = e = 1
3:    for i from 1 to n do
4:       for j from i to n do
5:          if (∑ p_i ⋯ p_j) > (∑ p_s ⋯ p_e)
6:            then s = i, e = j
7:          end if
8:       end for
9:     end for
10:   return (s, e)
11: end function
```

- $\Theta(n^3)$ time as written; can cache sums to achieve $\Theta(n^2)$

Divide-and-conquer (D&C) - brainstorm

- **Divide**: chop array in half into two smaller arrays L and R
- **Conquer**: recursively compute maximum subarray in L and in R
- **Combine**: choose a subarray of entire P that offers the maximum sum; the subarray P could be

1. subarray entirely in L;
2. subarray entirely in R; or
3. crossing subarray that starts in L and ends in R

- For Combine, options 1 and 2 are easy, option 3 is tricky

Identify crossing subarray - try 1

- Suppose the two pieces of P are P[*low* … *mid*] and P[*mid* + 1 … *high*]
- Tempting to try all pairs of $s \in \{low \ldots mid\}$ and $e \in \{mid + 1, \ldots, high\}$ would work, but
  - time becomes $T(n) = 2T(n/2) + \Theta(n^2)$ which is $\Theta(n^2)$ by master theorem
  - same time as brute force, but more complicated → not a win

Identify crossing subarray - try 2

- Not using the fact that a crossing subarray must cross *mid*: *s* is how far before *mid*; *separately*, *e* is how far after *mid*?
- two separate 1D searches → two linear loops: $\Theta(n) + \Theta(n) = \Theta(2n) = \Theta(n)$ time
- versus *s* is where, and *e* is how much later? → 2D search → two nested loops → $\Theta(n^2)$ time

Identify crossing subarray - try 2

```
1: function MAX-CROSSING-SUBARRAY(P, low, mid, high)
2:    leftsum = rightsum = - infinity
3:    sum = 0
4:    for i from mid down to low do
5:       sum = sum + P[i ]
6:       if sum > leftsum then
7:          leftsum = sum
8:          maxleft = i
9:       end if
10:   end for
11:   sum = 0
12:   for i from mid + 1 to high do
13:      sum = sum + P[i ]
14:      if sum > rightsum then
15:         rightsum = sum
16:         maxright = i
17:      end if
18:   end for
19:   return (maxleft; maxright; leftsum + rightsum)
20: end function
```

Maximum subarray algorithm – D&C solution

```
1: function MAX-SUBARRAY(P, low, high)
2:    if low == high then
3:       return (low, high, P[low])
4:    else
5:       mid = floor((low + high)/2)
6:       (leftlow, lefthigh, leftsum) = MAX-SUBARRAY(P, low, mid)
7:       (rightlow, righthigh, rightsum) = MAX-SUBARRAY(P, mid + 1, high)
8:       (crosslow, crosshigh, crosssum) = MAX-CROSSING-SUBARRAY(A, low, mid, high)
9:       if leftsum >= rightsum and leftsum >= crosssum then
10:         return (leftlow, lefthigh, leftsum) // entirely-left subarray
11:      else if rightsum >= leftsum and rightsum >= crosssum then
12:            return (rightlow, righthigh, rightsum) // entirely-right subarray
13:         else
14:            return (crosslow, crosshigh, crosssum) // mid-crossing subarray
15:         end if
16:      end if
17: end function
```

Time complexity of D&C solution

- D&C runtime is $T(n) = 2T(n/2) + \Theta(n)$
- Solves to $\Theta(n \log n)$, by master theorem, same as merge sort.
- D&C is much faster than brute force
- D&C benefits from observation that subarrays are contiguous, so extend in two directions from a middle, while brute force is oblivious to this

Best algorithm – Dynamic Programming

- Best algorithm is Kadane's algorithm and uses Dynamic Programming
- Read https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/
- IDEA:
  - look for all positive contiguous segments of the array (`max_ending_here` is used for this)
  - and keep track of maximum sum contiguous segment among all positive segments (`max_so_far` is used for this)
  - each time we get a positive sum compare it with `max_so_far` and update `max_so_far` if it is greater than `max_so_far`

Kadane's alg

(taken from https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/)

```
1: function DYN-PROG-MAX-SUBARRAY(P)
2:   max_so_far = INT_MIN
3:   max_ending_here = 0
4:   start = end = s = 0;
5:   for i from 1 to n do
6:      max_ending_here = max_ending_here + P[i]
7:      if (max_so_far < max_ending_here)  then
8:         max_so_far = max_ending_here
9:         start = s;
10:        end = i;
11:     endif
12:     if (max_ending_here < 0) then
13:        max_ending_here = 0
14:        s = i + 1;
15:    endif
16:  endfor
17: return (max_so_far, start, end)
18: end function
```

Decrease-by-half or Divide and Conquer

(taken from https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/03-divide-and-conquer.pdf)

One of the big ideas of computer science problem solving:

1. Divide a problem into smaller parts
2. Conquer the smaller problems recursively
3. Combine the smaller solutions into one solution for the original problem

(Ancient Roman politicians: divide your enemies (by getting them to distrust each other) and then conquer them one after another.)

Divide-and-conquer, outside of algorithm design

- Software design; breaking features into classes, functions
- Networking; OSI seven layer model
- Parallel processing; MapReduce
- Software process; agile methods; sprints

Divide-and-conquer pattern

```
1: function DIVIDE-AND-CONQUER(INPUT)
2:    if INPUT is base case then
3:       return trivial base case solution
4:    else
5:       x1, x2, …, xk = divide INPUT into k pieces (often 2)
6:       s1 = DIVIDE-AND-CONQUER(x1)
7:       ⋮
8:       sk = DIVIDE-AND-CONQUER(xk )
9:       S = combine s1, …, sk into one solution
10:      return S
11:   end if
12: end function
```

Time complexity

- When an algorithm contains a recursive call to itself, its running time can be expressed as a recurrence equation (aka recurrence)
- Recursive pseudocode leads to recurrences in run-time functions
- We denote by T(*n*) be the r.t. of a problem of size *n*.
- Suppose base case is n = 1 and takes Θ(1) time; in the recursive case we divide evenly into *b* pieces of size ≈ *n/b*, recurse once on *a* pieces out of the *b* pieces, and spend *f(n)* time in the divide and conquer steps:

$$T(n) = \begin{cases} \Theta(1) & if\ \mathrm{n}\ = 1 \\ aT(n/b) + f(n) & if\ \ n > 1 \end{cases}$$

Taking liberties with recurrence functions

- Algorithm analysis: ordinarily bounding asymptotically; Θ notation will hide constant factors anyway
- Drop math details that can only impact constants and add clutter
  - drop ceilings/floors, so write e.g. n/2 in lieu of $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil$
  - when the base case is Θ(1) time for n < c for some $c \in \Theta(1)$, don't bother writing it explicitly; so

$$T(n) = \begin{cases} \Theta(1) & if\ \mathrm{n}\ < c \\ aT(n/b) + f(n) & if\ \ n \geq c \end{cases}$$

is abbreviated as $T(n) = aT(n/b) + f(n)$

Master Theorem

- A function is a *master-form recurrence* when it may be defined as

$$T(n) = \begin{cases} \mathrm{O}(1) & \mathrm{n} < c \\ aT(n/b) + f(n) & n \geq c \end{cases}$$

Where f(n) is upper bounded by $\mathrm{O}(n^k)$ $f(n) \in \mathrm{O}(n^k)$, and $a, b, c, k \in \mathrm{O}(1)$ such that $c, a \geq 1, b > 1$, and $\mathrm{k} \geq 0$.

- A function is a *master-form recurrence* when it can also satisfy

$$T(n) \leq \begin{cases} \mathrm{O}(1) & \mathrm{n} < c \\ aT(n/b) + f(n) & n \geq c \end{cases}$$

- If T is a *master-form recurrence* function, then

$$T(n) = \begin{cases} O(n^k) & if\ a < b^k \quad (case\ 1) \\ O(n^k \log_2 n) & if\ a = b^k \quad (case\ 2), or \\ O(n^{\log_b a}) & if\ a > b^k \quad (case\ 3). \end{cases}$$

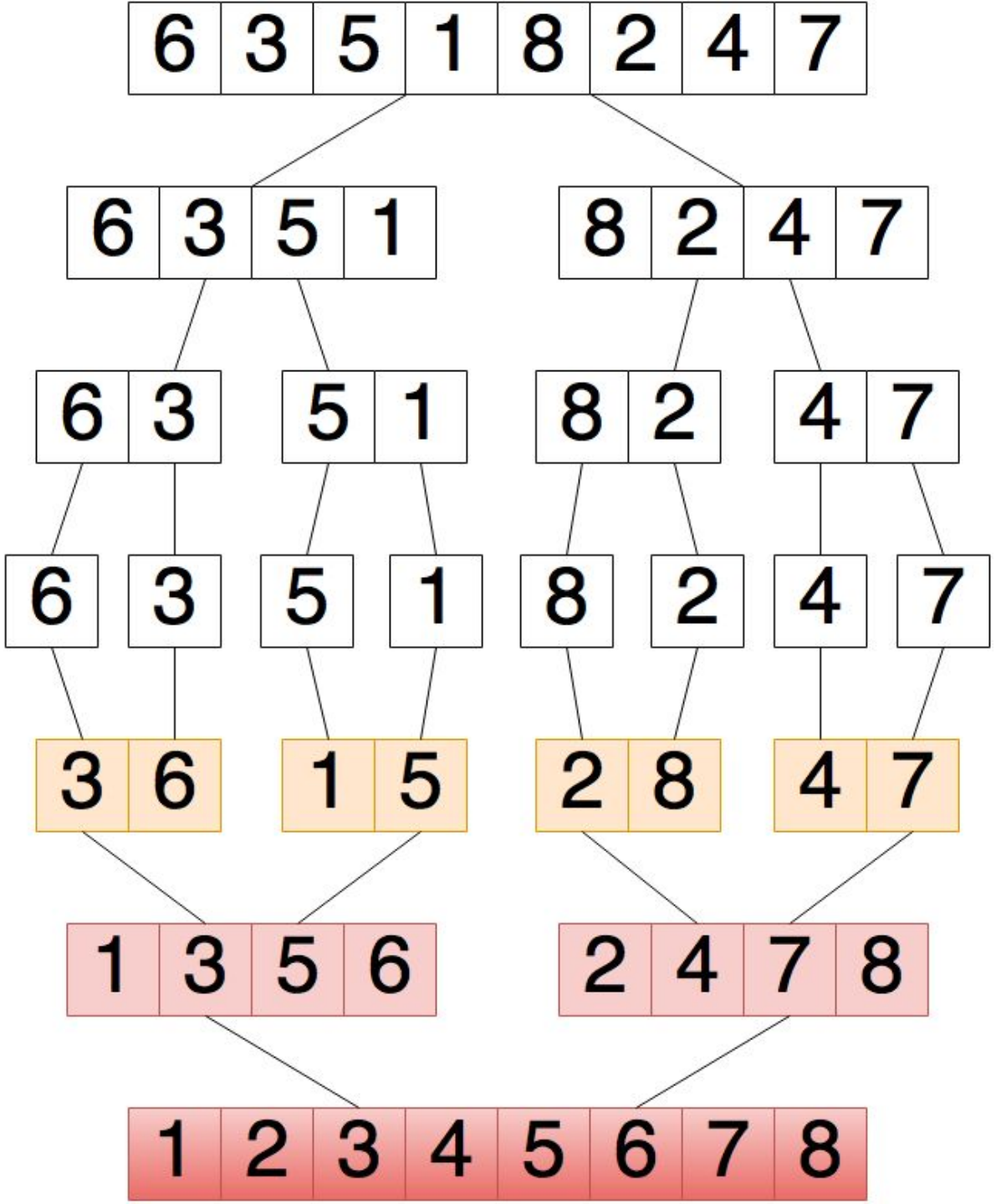
- Reaching the results is done by solving inhomogeneous recurrent equations
- Use the master theorem to compute the time complexity for mergesort and binary search.

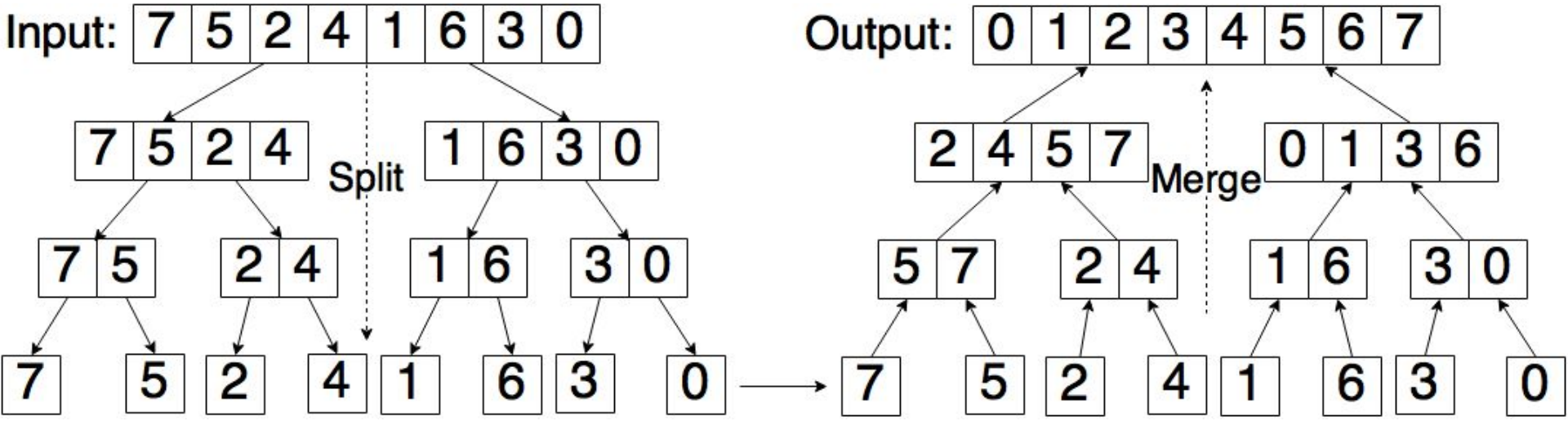
Merge Sort

- Merge sort is a divide-and-conquer algorithm that was invented by John von Neumann in 1945
- Given a vector of *n* comparable elements V, stored in an array V[1 ...n], sort the sequence in non-descending order; Objective: the sorted sequence S is a permutation S of V
- Follow the divide-and-conquer paradigm for sorting:

1. Divide: Split V down the middle into two subsequences, each of size roughly *n*/2.
2. Conquer: Sort each subsequence (by calling `merge_sort` recursively on each).
3. Combine: Merge the two sorted subsequences into a single sorted list (by calling `merge`).

```
Merge-Sort(A,i,j)
  if (i<j) then
      m = (i+j)/2
      Merge-Sort(A,i,m)
      Merge-Sort(A,m+1,j)
      merge(A,i,m,j)
```


6 3 5 1 8 2 4 7
6 3 5 1
8 2 4 7
6 3
5 1
8 2
4 7
6
3
5
1
8
2
4
7
3 6
1 5
2 8
4 7
1 3 5 6
2 4 7 8
1 2 3 4 5 6 7 8



- Pseudocode:

```
MergeSort(array V, int p, int r) {
   if (p == r) then // do nothing
   else {
      // we have at least 2 items
      q = (p + r)/2
      MergeSort(V, p, q) // sort V[p..q]
      MergeSort(V, q+1, r) // sort V[q+1..r]
      Merge(V, p, q, r) // merge everything together
   }
}
```

Procedure Merge is described on the next slide.

```
Merge(array V, int p, int q, int r)  {
// merges V[p..q] with V[q+1..r]
// declare a temporary array S to store the sorted sequence
array S[p..r]
i = k = p // initialize pointers
j = q+1
while (i <= q and j <= r) {
   // while both subarrays are nonempty
   if (V[i] <= V[j]) then
     // copy from left subarray
     S[k++] = V[i++]
   else
     // copy from right subarray
     S[k++] = V[j++]
}
// copy any leftover to B from the left subarray
while (i <= q) do
   S[k++] = V[i++]
// copy any leftover to B from the right subarray
while (j <= r)  do
   S[k++] = V[j++]
// copy B back to A
for i = p to r do V[i] = S[i]
}
```

Observe that of the last two while-loops in the Merge procedure, only one will be executed. Why?

```
MergeSort(array V, int p, int r) {
   if (p == r) then // do nothing
   else {
      // we have at least 2 items
      q = (p + r)/2
      MergeSort(V, p, q) // sort V[p..q]
      MergeSort(V, q+1, r) // sort V[q+1..r]
      Merge(V, p, q, r) // merge everything together
   }
}
```

```
Merge(array V, int p, int q, int r)  {
// merges V[p..q] with V[q+1..r]
// declare a temporary array S to store
// the sorted sequence
array S[p..r]
i = k = p // initialize pointers
j = q+1
while (i <= q and j <= r) {
   // while both subarrays are nonempty
   if (V[i] <= V[j]) then
     // copy from left subarray
     S[k++] = V[i++]
   else
     // copy from right subarray
     S[k++] = V[j++]
}
// copy any leftover to B from the left subarray
while (i <= q) do
   S[k++] = V[i++]
// copy any leftover to B from the right subarray
while (j <= r)  do
   S[k++] = V[j++]
// copy B back to A
for i = p to r do V[i] = S[i]
}
```

Merge Sort is a Stable Sorting Algorithm

- Suppose that instead of the if-statement “if (V[i] <= V[j])”, we have the if-statement “if (V[i] < V[j])” .
  - In this case, if (V[i]==V[j]) we would copy from the right subvector.
- It wouldn’t have mattered for equal elements, if instead of copying from the left subvector we had copied from the right subvector; since the elements are equal, they can appear in either order in the final subvector.
- The reason to prefer this particular choice:
  - Many times we are sorting data that does not have a single attribute, but has many attributes (name, SSN, grade, etc.) Often the list may already have been sorted on one attribute (say, name). If we sort on a second attribute (say, grade), then it would be nice if people with same grade are still sorted by name.
- A sorting algorithm that has the property that equal items will appear in the final sorted list in the same relative order that they appeared in the initial input is called a *stable sorting* algorithm.
- By favoring elements from the left sublist over the right, we preserve the relative order of elements.
- It can be shown by induction that Merge Sort is a stable sorting algorithm.

Binary Search

- *Binary search* is an algorithm for searching for an element within an ordered list of comparable elements.
- The *binary search problem* is:

input: a vector V of n comparable elements in non-decreasing order, and a query value q

output: the index i such that V[i] = q, or None if no such index exists

- It is related to sequential search: the input is a query value q and a vector V that may or may not contain q.
- Binary search is only applied to lists that are already sorted, and that property allows for an algorithm that is faster than sequential search:
  - Sequential search runs in $O(n)$ time
  - The time complexity of Binary-Search(V, 1, n, q) is $O(\log n)$

- Follows the divide-and-conquer paradigm
  - Divide: the n-element vector V[i..j], i=1 and j=n, into two subsequence of n/2 length, V[i..m] and V[m+1..j] where m=(i+j)/2
  - Conquer: compare the middle element of the sequence with q and search the correct subsequence out of the two using binary-search
  - Combine: nothing

```
Binary-Search(V, i, j, q)
  if (i == j) then
    return (V[i] == q)
  else
    m = (i+j)/2
    if ( V[m] == q ) then
      return True
    else
      if ( q < V[m] ) then
        return Binary-Search(V,i,m-1,q)
      else
      // ( q > V[m] )
        return Binary-Search(V,m+1,j,q)
```

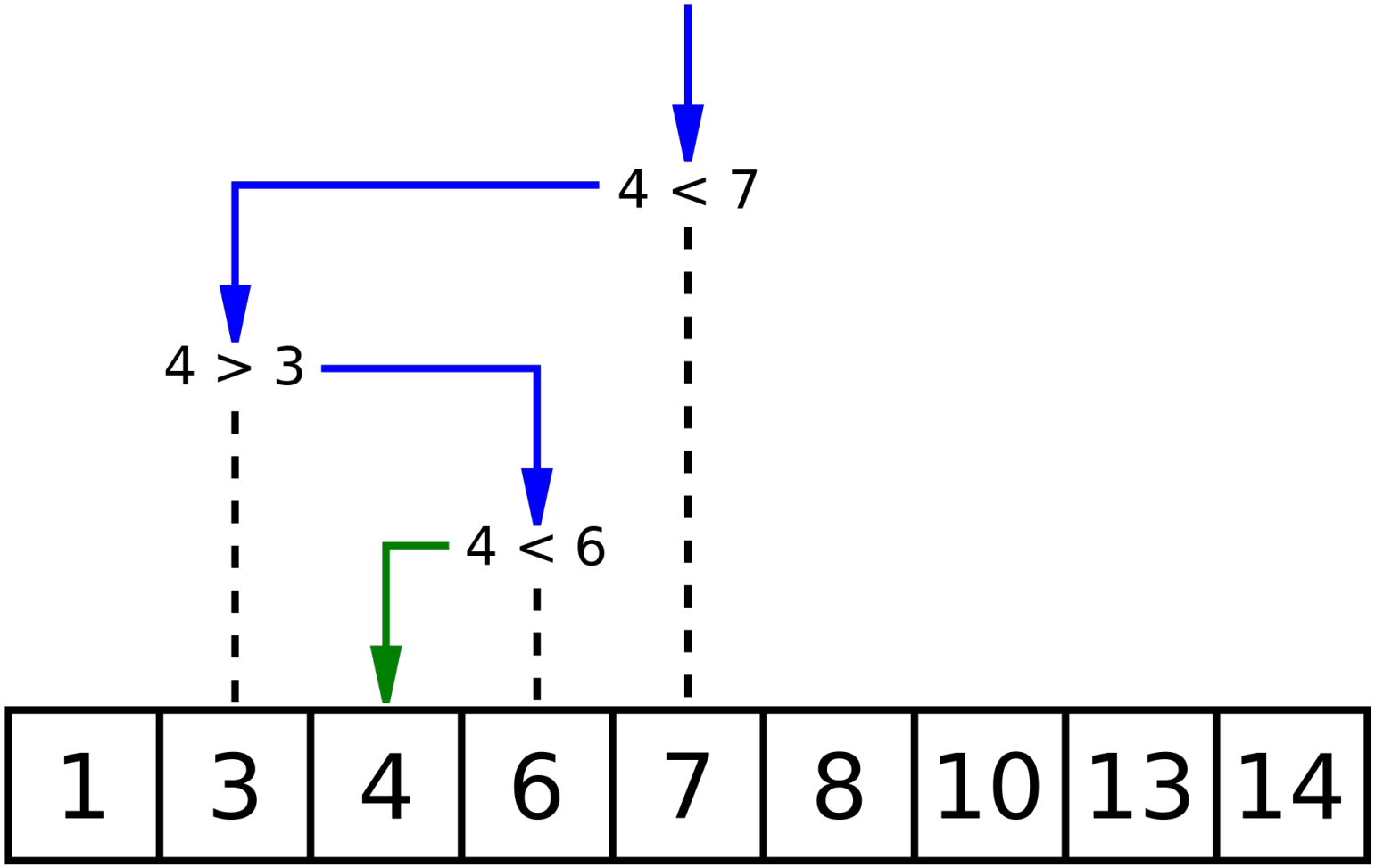


Image taken from wikipedia

Time Complexity of Mergesort

- The time complexity of MergeSort procedure when called with input *n* and *L* is O(n log n).
- Proof: We use the master theorem.
- The running time of the Merge procedure is 2n+3, thus O(n).
- Let $T(n)$ be the running time of MergeSort procedure when called with input *n* and *L*
  - For $n \leq 1$, then $T(n) = 2$.
  - For $n > 1$, then $T(n) = 1 + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2n + 3 \leq 2 \cdot T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 2n + 3.$

- Thus the function $T(n)$ is a *master-form recurrence*

$$T(n) \leq \begin{cases} 2 & n < 2 \\ 2T(n/2) + 2n + 3 & n \geq 2 \end{cases}$$

$2n + 3 \in O(n)$, thus k=1

$c = 2$

$a = 2$

$b = 2$

$k = 1$

All $t, r, d, k \in O(1)$.

Since $a = b^k$ (case 2), we apply the master theorem and we obtain $O\left(n^k \log n\right) = O(n \log n)$.

Time Complexity of Binary Search

- Thus the function $T(n)$ is a *master-form recurrence*

$$T(n) \leq \begin{cases} 2 & \text{n} < 2 \\ 2T(n/2) + 2n + 3 & n \geq 2 \end{cases}$$

$2n + 3 \in O(n)$, thus k=1

$c = 2$

$a = 2$

$b = 2$

$k = 1$

All $t, r, d, k \in O(1)$.

Since $a = b^k$ (case 2), we apply the master theorem and we obtain $O\left(n^k \log n\right) = O(n \log n)$.

Powering a number

(https://users.cs.duke.edu/~reif/courses/alglectures/demleis.lectures/lec3.pdf)

- Compute a^n when $n \in \mathbb{N}$ (the set of all natural numbers).

- Naive algorithm: O(n), pseudocode below:

```
P =1
for i=1 to n do
  P = P *a
return(P)
```

- Divide-and-conquer algorithm: O(log n), pseudocode next slide.

Powering a number (contd.)

- We use the following relation

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

- By the master theorem: T(n) = T(n/2) + O(1) => T(n) = O(log n).

```
DaC(a,n)
  if n ==0) return(1)
  elsif (n==1) return(a)
  elsif (n%2 == 0) // even number
    B = DaC(a,n/2)
    return(B*B)
  else // odd number
    B = DaC(a,(n-1)/2)
    return(B*B*a)
```

Computing Fibonacci numbers

(https://users.cs.duke.edu/~reif/courses/alglectures/demleis.lectures/lec3.pdf)

Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

0 1 1 2 3 5 8 13 21 34 ⋯

- Naïve recursive algorithm: exponential time
- Use Binet's formula (https://en.wikipedia.org/wiki/Fibonacci_number) :

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} \text{ where } \varphi = \frac{1+\sqrt{5}}{2} \text{ is the } \textit{golden ratio}.$$

$\sqrt{5}$ is an irrational number and computers cannot represent irrationals, so F_n will be approximated

Computing Fibonacci numbers (contd.)

(https://users.cs.duke.edu/~reif/courses/alglectures/demleis.lectures/lec3.pdf)

- Dynamic programming algorithm: linear time, O(n), by computing F[0], F[1], … F[n] as the sum of the previously two computed Fibonacci numbers.
- Divide-and-conquer using recursive squaring of matrices: logarithmic time, O(log n); two methods:
  - Naïve recursive squaring: compute $F_n = \phi^n/\sqrt{2}$ rounded to the nearest integer. Unreliable method since floating-point arithmetic is prone to round-off errors
  - Matrix recursive squaring: Use the theorem $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$ to compute the 2x2 matrix.

Matrix Multiplication

- Given two matrices $A=[a_{ij}]$ and $B=[b_{ij}]$, compute $C= [c_{ij}] =A \cdot B$, with i,j=1,..n.

Input: $A = [a_{ij}], B = [b_{ij}].$
Output: $C = [c_{ij}] = A \cdot B.$
$\Big\}\ i, j = 1, 2, \ldots, n.$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Matrix Multiplication (contd.)

- Algorithm learnt in school:

```
for i=1 to n do
  for j = 1 to n do
    c[i][j] = 0
    for k=1 to n do
      c[i][j] = c[i][j] + a[i][k] * b[k][j]
```

- Time complexity: $O(n^3)$

Matrix Multiplication (contd.)

- Divide-and-conquer: the n×n matrix is equivalent to a 2×2 matrix, each element being an (n/2)×(n/2) submatrix

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C = A \cdot B$$

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dg \\ u &= cf + dh \end{aligned}\right\} \begin{array}{l} 8 \text{ mults of } (n/2)\times(n/2) \text{ submatrices} \\ 4 \text{ adds of } (n/2)\times(n/2) \text{ submatrices} \end{array}$$

mults = multiplications
adds = additions

Matrix Multiplication (contd.)

- Divide-and-conquer: the n×n matrix is equivalent to a 2×2 matrix, each element being an (n/2)×(n/2) submatrix

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C = A \cdot B$$

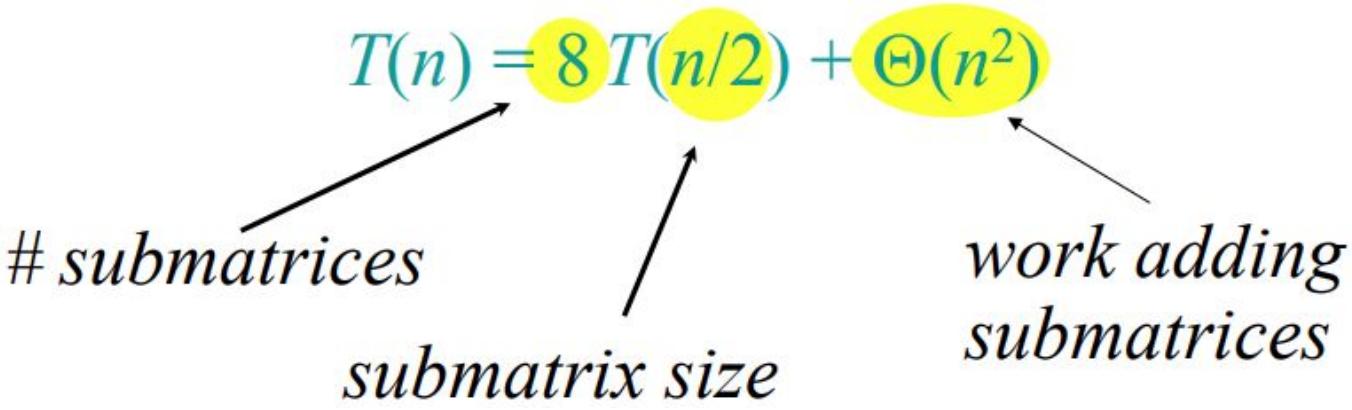
$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dh \\ u &= cf + dg \end{aligned}\right\}$$

recursive

8 mults of $(n/2)\times(n/2)$ submatrices

4 adds of $(n/2)\times(n/2)$ submatrices

Matrix Multiplication (contd.)



$t = 2$

$r = 8$

$d = 2$

$k = 2$

All $t, r, d, k \in$ O(1). Since $r > d^k$ (case 3), we apply the master theorem and we obtain $T(n) \epsilon O\left(n^{\log_d r}\right) = O(n^3)$.

No better than the first algorithm.

Matrix multiplication: Strassen idea

- Strassen idea: Multiply 2×2 matrices using only 7 recursive mults.
- Let A and B be the two input matrices, 2x2
- Let C = A · B, so we would have to have

r = a · e + b · g

s = a · f + b · h

t = c · e+d · g

u = c · f+d · h

which is the standard matrix multiplication.

- Instead, we consider 7 new terms:

P1 = a · (f-h)

P2 = (a+b) · h

P3 = (c+d) · e

P4 = d · (g-e)

P5 = (a+d) · (e+h)

P6 = (b-d) · (g+h)

P7 = (a-e) · (e+f)

and we now have r = P5 + P4 - P2 + P6

s = P1 + P2

t = P3 + P4

u = P5 + P1 - P3 - P7

with a total of **7 mults and 18 adds/subs**

Strassen's algorithm for any matrices

- It can be generalized to any pair of n×n matrices A and B

1. ***Divide:*** Partition A and B into $(n/2)\times(n/2)$ submatrices. Form terms to be multiplied using + and – .
2. ***Conquer:*** Perform 7 multiplications of $(n/2)\times(n/2)$ submatrices recursively.
3. ***Combine:*** Form C using + and – on $(n/2)\times(n/2)$ submatrices.

- Time complexity: $T(n) = 7T(n/2) + O(n^2)$

Strassen's algorithm for any matrices (contd.)

- Time complexity: $T(n) = 7T(n/2) + O(n^2)$

$t = 2$

$r = 7$

$d = 2$

$k = 2$

All $t, r, d, k \in O(1)$. Since $r > d^k$ (case 3), we apply the master theorem and we obtain $T(n) \epsilon O\left(n^{\log_d r}\right) = O\left(n^{\log_2 7}\right)$.

- The number $\log_2 7$ =2.81.. Is almost 3 but because the difference is in the exponent, the impact on running time is significant.
- Best to date, of theoretical interest only $O(n^{2.3728\ldots})$

Counting paths between two vertices in a graph

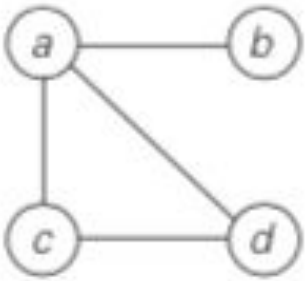
(taken from Levitin, page 242)

- It is not difficult to prove by mathematical induction that the number of different paths of length $k > 0$ from the ith vertex to the jth vertex of a graph (undirected or directed) equals the (i, j)th element of A^k where A is the adjacency matrix of the graph.
- Therefore, the problem of counting a graph's paths can be solved with an algorithm for computing an appropriate power of its adjacency matrix.
- Note that the exponentiation algorithms we discussed before for computing powers of numbers are applicable to matrices as well.

Example

(taken from Levitin, page 242)

- Consider the graph of Figure 6.16.
- Its adjacency matrix A and its square A^2 indicate the numbers of paths of length 1 and 2, respectively, between the corresponding vertices of the graph.
- There are three paths of length 2 that start and end at vertex a, (a − b − a, a − c − a, and a − d − a), but there is only one path of length 2 from a to c, (a − d − c).



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array} \qquad A^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \end{array}$$

FIGURE 6.16 A graph, its adjacency matrix A, and its square A^2. The elements of A and A^2 indicate the numbers of paths of lengths 1 and 2, respectively.

Indivisible Problems

- The decrease-by-a-fraction pattern cannot be used to solve all problems.
- The viability of the pattern depends on two properties of the problem at hand:

1. It must always be possible to divide any instance into two sub-instances of roughly equal size; and
2. It must always be possible to combine the solutions to those sub-instances into a correct output for the entire original instance.

Example of indivisible problems

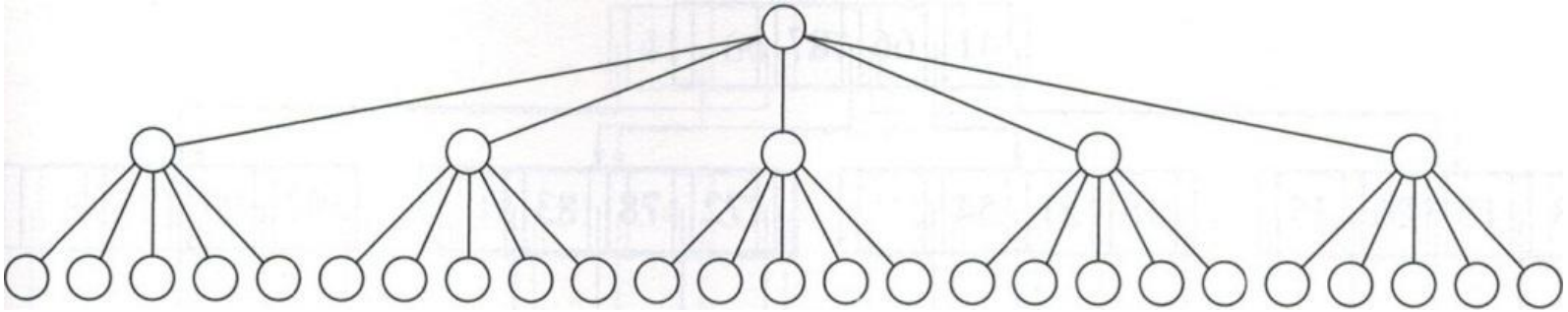
- For example, it is not always possible to divide a graph with n vertices and m edges into two smaller subgraphs with $\approx n/2$ vertices and $\approx m/2$ edges, since edges need not be distributed evenly throughout the graph, and some of the graph's edges may straddle the boundary between the two subgraphs.
  - (As a matter of fact, actually there are algorithms for finding separators of graphs for use with the decrease-by-a-fraction pattern, but they are beyond the scope of this course)
  - A good example is the traveling salesmen problem
- A problem's structure may make it impossible to compute decrease_by_half(L) and decrease_by_half(R) independently of one another.
  - Some problems involve "global" interdependencies that necessitate processing an entire instance all at once rather than piecemeal.
  - A good example is the circuit satisfaction problem.

B-Trees

- Let us assume that we have to search for an item along a very large number of items (about 10 million items); we are concerned only with keys
- A balanced binary tree would have a depth of $\lceil log 10^7 \rceil \approx 24$
- How about if we branch more, i.e. we allow a node to have more than two children?
- More branching means less height:
  - A perfect binary tree of 31 nodes has five levels
  - A 5-ary tree of 31 nodes has only three levels
- A *balanced binary tree* has the minimum possible height (a.k.a. maximum depth for the leaf nodes), because for any given number of nodes, the leaf nodes are placed at the greatest possible depth.

Motivation for B-trees and B^+-trees

- A *B-tree* is a generalization of a binary search tree in that a node can have more than two children.
  - A B-tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in log arithmetic time.
- The major drawback of B-tree is the difficulty of traversing the keys sequentially.
- A *B+ tree* retains the rapid random-access property of the B-tree while also allowing rapid sequential access.
  - Each node of the tree contains an ordered list of keys and pointers to lower-level nodes in the tree. These pointers can be thought of as being between each of the keys.
- A B+ tree is a data structure often used in the implementation of database indexes.



- If *N* is the total number of nodes
  - A complete binary tree would have a depth of $\lceil logN \rceil$
  - An M-ary tree would have a depth of $\lceil log_M N \rceil$
- An *M*-ary search tree can be created like a binary search tree:
  - In a binary search tree, we need *one* key to decide which branch out of the two branches to take
  - In a *M*-ary search tree, we need $(M-1)$ keys which branch out of the *M* branches to take
- We need to ensure that the *M*-tree is balanced otherwise it could degenerate into a linked list or a binary search tree
- B-trees are balanced search trees designed to work well on magnetic disks or other direct-access secondary storage device and they minimize the disk I/O operations.

B-tree

- Introduced by Bayer and McCreight in 1972
- There are two distinct definitions of what *order of a B-tree* means: order can mean the *t*=minimum number of **children** in a non-root node (Bayer and McCreight's definition, textbook) or order can mean the *m*=maximum number of children (Knuth)
- Both definitions are compatible with each other; we have that $t = \lceil m/2 \rceil$ but be careful when using one definition versus the other one when defining the B-tree
- For example, B-tree with order m=4 (4-ary tree) can be a B-tree with t=2, while a B-tree with t=2 may correspond to a B-tree of order m=3 (3-ary tree) or a B-tree of order m=4 (4-ary tree)
- The simplest B-tree occurs when t=2. Every internal node then has either 2, 3, or 4 children, and we have a 2-3-4 tree. (Do not confuse a 2-3-4 tree with a 4-ary tree.)

B-tree Definition (Knuth)

- Knuth's Definition^^: A ***B-tree*** of order *m* (*m*-ary tree) is a tree with the following properties:

1. Every node has at most *m* children
2. All internal nodes except the root have at least $\lceil m/2 \rceil$ children
3. Every non-leaf node has at least two children
4. All leaves appear on the same level and carry no information.
5. A non-leaf node with k children contains k−1 keys

And these keys partition the keys in the children in the fashion of a search tree. Key *i* stored at some non-leaf node represents the smallest key in subtree *i* + 1.

^^ https://en.wikipedia.org/wiki/B-tree#:~:text=In%20computer%20science%2C%20a%20B,with%20more%20than%20two%20children.

B-tree Definition (Kunth)

- In Kunth's definition, *internal* nodes (also known as inner nodes) are all nodes except for leaf nodes and the root node.
- The *root* node has the same upper limit on the number of children as internal nodes but has no lower limit. For example, when there are fewer than L−1 elements in the entire tree, the root will be the only node in the tree with no children at all.
- The *leaf* nodes do not carry any information.
- The internal nodes that are one level above the leaves are what would be called "leaves" by other authors: these nodes only store keys (at most m-1, and at least m/2-1 if they are not the root) and pointers to nodes carrying no information.

B-tree (Bayer and McCreight, textbook)

- Definition based on the minimum degree (from the textbook): A ***B-tree*** is a rooted tree with the following properties:

1. Each node has several keys, stored in non-decreasing order from left to right
2. For an internal node, the number of children is equal to 1 + number of keys stored at a node
3. Each key separates the ranges of keys stored in each subtree
4. All leaves have the same depth which is the tree's height
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer t > 1 called *the minimum degree* of the B-tree:

B-tree (Bayer and McCreight, textbook)

- (def, continue):

a. Every node other than the root must have at least t - 1 keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

b. Every node may contain at most 2t - 1 keys. Therefore, an internal node may have at most 2t children. We say that a node is *full* if it contains exactly 2t-1 keys.

Bayer and McCreight considered the leaf level to be the lowest level of keys, so leaf nodes carry keys.

Restriction on the height

(taken from https://www.geeksforgeeks.org/introduction-of-b-tree-2/)

- *The minimum height of the B-Tree that can exist with n number of nodes and m is the maximum number of children of a node can have is:*

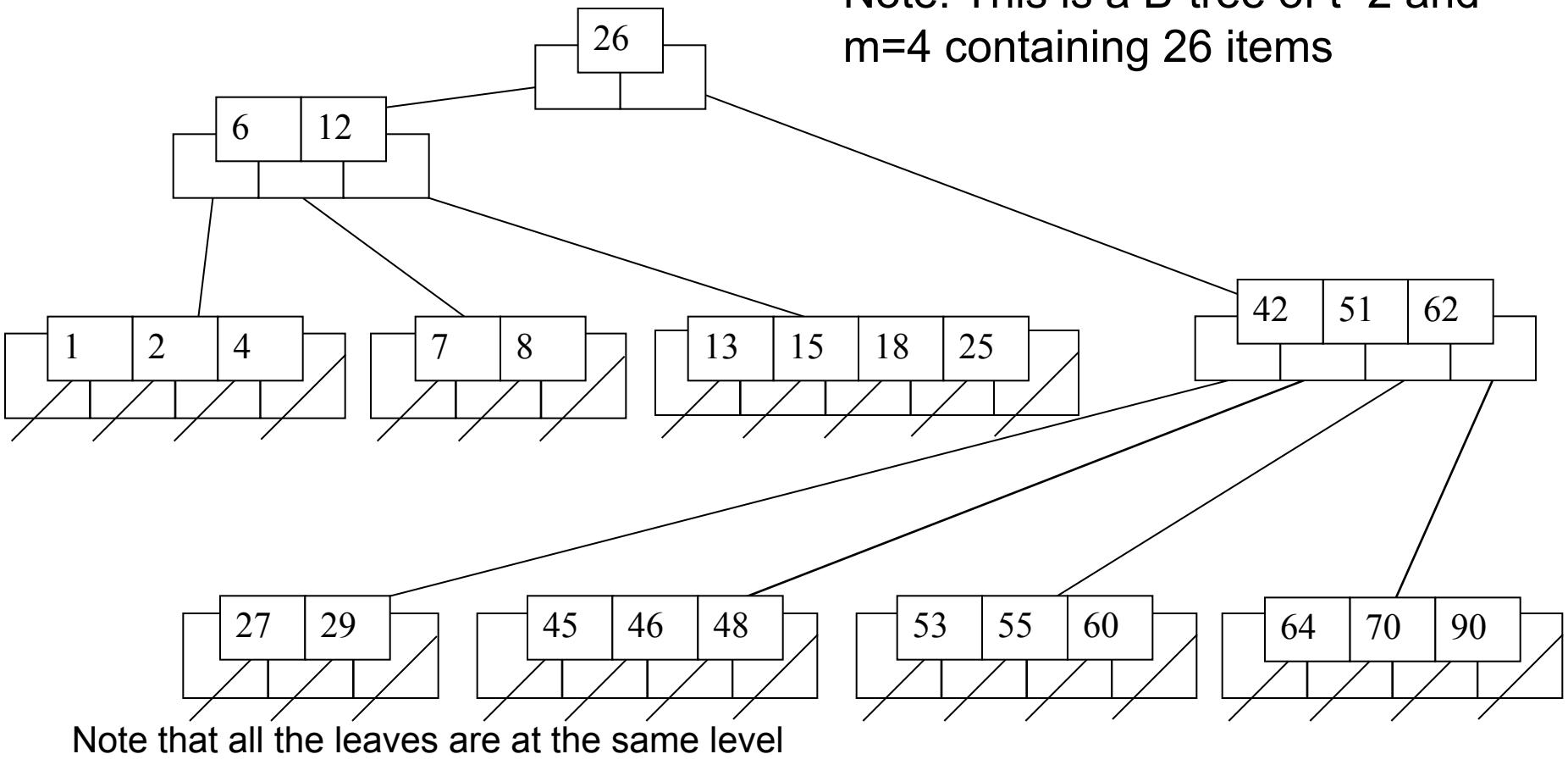
$$h_{min} = \lceil \log_m (n+1) \rceil - 1$$

- *The maximum height of the B-Tree that can exist with n number of nodes and t is the minimum number of children that a non-root node can have is:*

$$h_{max} = \lfloor \log_t \frac{n+1}{2} \rfloor \text{ and } t = \lceil \frac{m}{2} \rceil$$

An example B-Tree

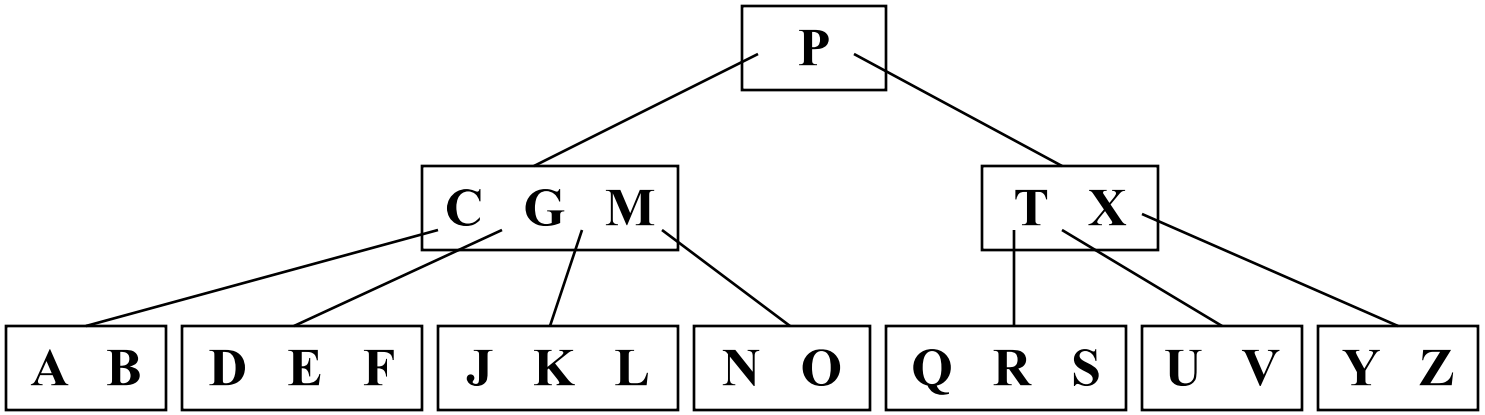
(taken from people.cs.pitt.edu/~injungkim/1004_lab_Btree)



Example of B-trees

(slides taken from http://people.cs.aau.dk/~simas/aalg04/)

- Let's draw a B-tree of t=2 and m=4 with the keys in the range A through Z
- There are a total of 23 keys



B-tree operations

- An implementation needs to support the following B-tree operations (corresponds to Dictionary ADT operations):
  - **Search** (simple)
  - **Create** an empty tree (trivial)
  - **Insert** (complex): depends on split and merge
  - **Delete** (complex): depends on merge