COT 6405 Introduction to Theory of Algorithms

Midterm I review

Coverage

- Midterm I will cover everything we have learned so far (quicksort is not included)
  - From Intro lecture to Lecture 7 (inclusive)
  - Function growth rate analysis, divide and conquer, recurrence, recursion tree and the Master Theorem, heaps, basic heap operations, and heapsort

Exam policy

- Closed books, closed computers, and closed notes.
- Calculators (not smart phones and laptops) are allowed
- Location: ENB 118 (regular session students)
  ENB 313 (online session students)

Preliminaries (cont'd)

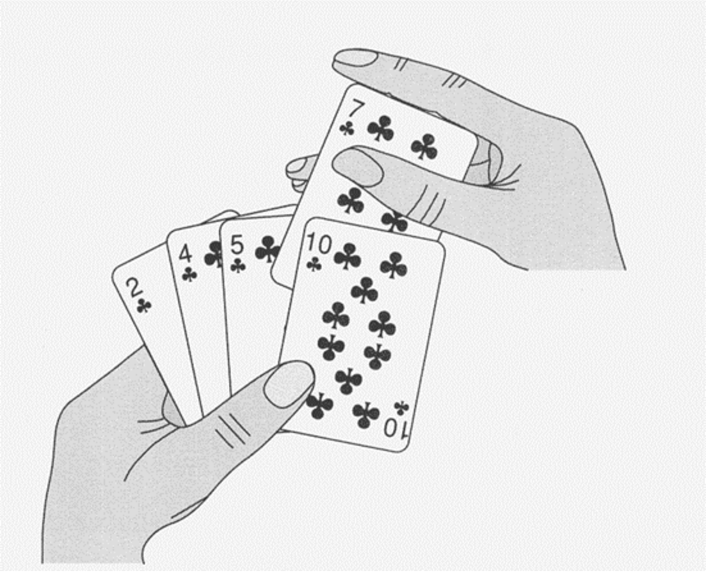
- $\log_a n = \frac{\log_b n}{\log_b a}$
- $x^{\log_a y} = y^{\log_a x}$
- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
- $\sum_{i=0}^{t} r^i = \frac{1-r^{t+1}}{1-r} \; if \; r \neq 1$
- $\sum_{i=0}^{k} i a^i = \frac{a(1-a^k)}{(1-a)^2} - \frac{k a^{k+1}}{1-a}$
- $\sum_{i=0}^{\infty} i a^i = \frac{a}{(1-a)^2}$

What is an Algorithm?

- A well defined computational procedure that
  - Takes some values as input and produces some values as an output.
- A tool for solving a well-specified computer problem
- A strategy to solve a problem.
  - E.g., how to find students that have the same birthdays in this classroom?

Insertion Sort

```
for j = 2 to n {
        key = A[j];
        i = j -1;
        While (i > 0) and (A[i] > key) {
            A[i+1] = A[i];
            i = i – 1;
        }
        A[i+1] = key;
    }
```



Growth "classes" of functions

- $O(g(n))$ big oh: upper bound on the growth rate of a function;
  - That is, a function belongs to class $O(g(n))$ if $g(n)$ is an upper bound on its growth rate
- $\Omega(g(n))$ big omega: lower bound on the growth rate of a function
- $\Theta(g(n))$ big theta: exact bound on the growth rate of a function

Precise definitions of big oh and big omega

- f(n) $\in$ O(g(n)) iff there exist c > 0 and n_0> 0 such that f(n) $\leq$ cg(n) for all n $\geq n_0$
- f(n) $\in$ Ω(g(n)) iff there exist c > 0 and n_0 > 0 such that f(n) $\geq$ cg(n) for all n $\geq n_0$
- Θ (g(n) $\in$ O(g(n)) $\cap$ Ω(g(n))

Limits and notation

- Limits can be helpful in determining the growth rate of functions
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ = 0 implies $f(n) \in$ o(g(n)), that is, $f(n) \notin \Omega$(g(n))
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ = ∞ implies $f(n) \in \omega$(g(n)), that is, $f(n) \notin O$(g(n))
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ = $d > 0$ implies $f(n) \in \Theta$(g(n))

Asymptotic proofs

- Ex1: Prove $n^3 - 10n^2 \notin O(n^2)$
- Ex2: Prove $5n^3 - 3n^2 + 2n - 6 \in \Theta(n^3)$

Divide and Conquer

To solve (an instance of) a problem P

IF (the instance of) P is “large enough” THEN

Divide P into smaller instances of the same problem

Recurse to solve the smaller instances

Combine solutions of the smaller instances to create a solution for the original instance

ELSE

Solve P directly

Merge sort

- The *merge sort* algorithm closely follows the divide-and-conquer paradigm
  - **Divide:** divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
  - **Conquer:** sort the two subsequences recursively using merge sort
  - **Combine:** merge the two sorted subsequences to produce the sorted answer

The recursion “bottoms out” when the sequence to be sorted has length 1

Merge sort (cont'd)

MERGE-SORT (A, p, r)

if p < r

q = $\lfloor (p + r)/2 \rfloor$

MERGE-SORT (A, p, q)

MERGE-SORT (A, q+1, r)

MERGE (A, p, q, r)

Merging

MERGE (A, p, q, r)

$n_1 = q - p + 1$; $n_2 = r - q$

create arrays $L[1 \,..\, (n_1 + 1)]$ and $R[1 \,..\, (n_2 + 1)]$

for $i = 1$ **to** n_1

  $L[i] = A[p + i - 1]$

for $j = 1$ **to** n_2

  $R[j] = A[q + j]$

$L[n_1 + 1] = \infty$; $R[n_2 + 1] = \infty$; $i = 1$; $j = 1$;

for $k = p$ **to** r

  if $L[i] \leq R[j]$

    then $A[k] = L[i]$

      $i = i + 1$

  else $A[k] = R[j]$

    $j = j + 1$

Recurrences

- What is a recurrence?
  - An equation that describes a function in terms of its value on smaller functions
- The time complexity of divide-and-conquer algorithms can be expressed as recurrences

Recurrence Examples

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

Solving the recurrences

- Substitution method
- Recursion Tree
- Master method

Substitution method

- The substitution method comprises two steps:
  - 1. Guess the form of the solution
  - 2. Use mathematical induction to show the correctness of the guess

Avoiding Pitfalls

- It is easy to err in the use of asymptotic notation
- Solve $T(n) = 2T(n/2) + n$
- Guess: $T(n) = O(n)$ and $T(k) \leq ck$ for all k < n and some positive constant number c
- Induction: $T(n) \leq 2(c(n/2)) + n$

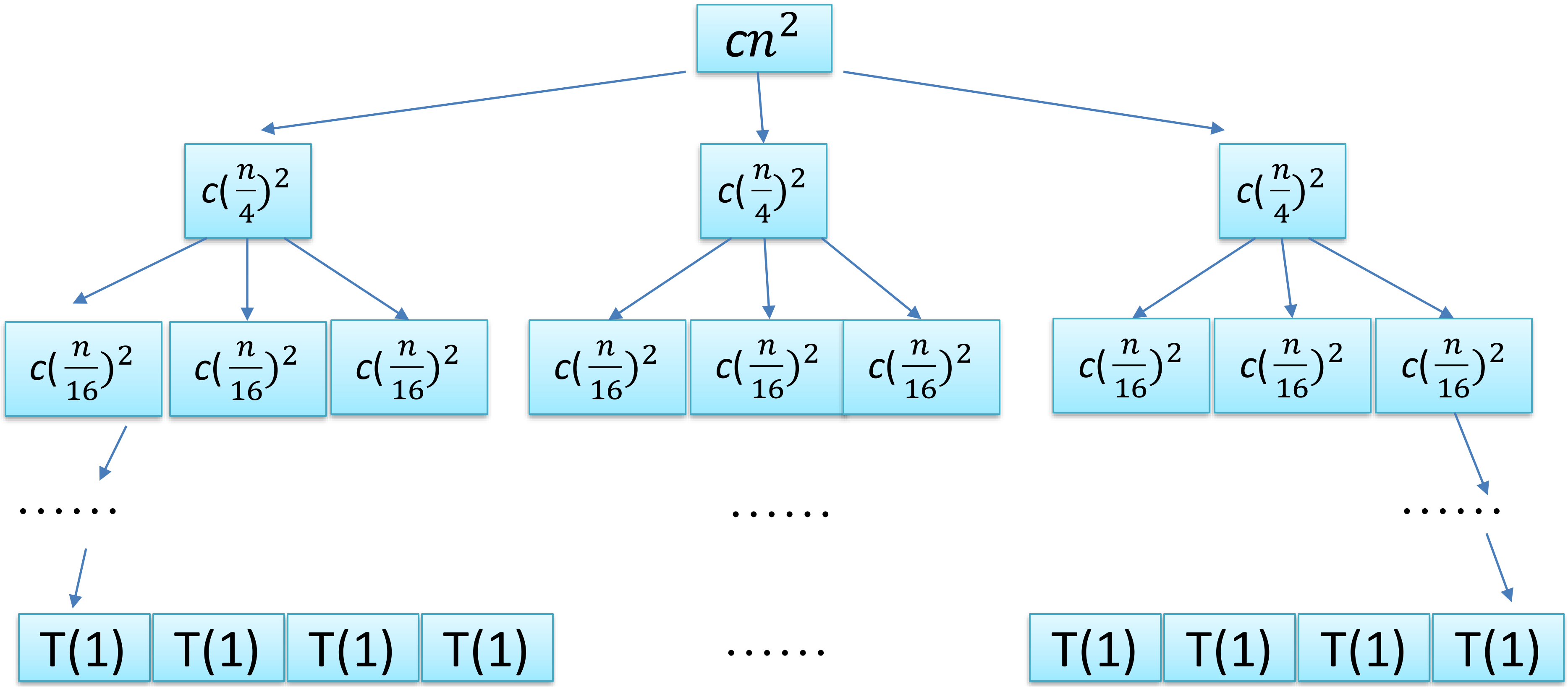
$$\leq cn + n = O(n)$$

Recursion tree method (cont'd)

- An alternative approach: draw a tree to diagram all the recursive calls that take place
- Solve T(n) = 3T(n/4) + cn^2

Example

- Solve $T(n) = 3T(n/4) + cn^2$



The Master Theorem (Cont'd)

- If T(n) = aT(n/b) + f(n) then

$\boxed{f(n) < n^{\log_b a}}$

$$
\boldsymbol{T(n)} = \begin{cases} \Theta\left(\boldsymbol{n}^{\log_b \boldsymbol{a}}\right) & \boldsymbol{f(n)} = \boldsymbol{O}\left(\boldsymbol{n}^{\log_b \boldsymbol{a}-\varepsilon}\right) \\ \Theta\left(\boldsymbol{n}^{\log_b \boldsymbol{a}} \ \mathbf{lg}^{\boldsymbol{k}+1} \boldsymbol{n}\right) & \boldsymbol{f(n)} = \Theta\left(\boldsymbol{n}^{\log_b \boldsymbol{a}} \ \mathbf{lg}^{\boldsymbol{k}} \boldsymbol{n}\right) \\ \Theta\left(\boldsymbol{f(n)}\right) & \boldsymbol{f(n)} = \Omega\left(\boldsymbol{n}^{\log_b \boldsymbol{a}+\varepsilon}\right) \text{AND} \\ & \boldsymbol{af(n/b)} < \boldsymbol{cf(n)} \ \text{for large } \boldsymbol{n} \end{cases} \left. \vphantom{\begin{matrix} a \\ b \\ c \\ d \end{matrix}} \right\} \begin{matrix} \varepsilon > 0 \\ \boldsymbol{c} < 1 \end{matrix}
$$

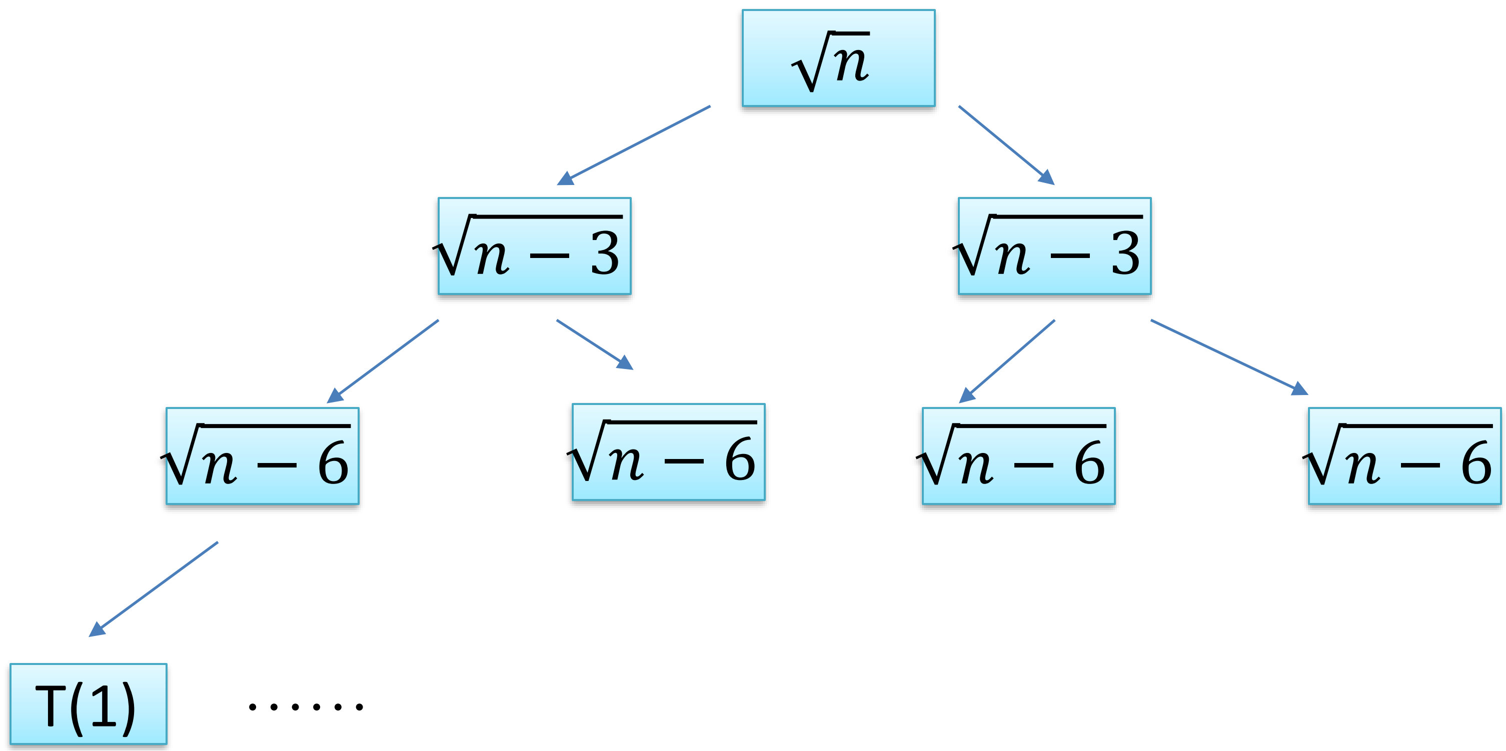
$\boxed{f(n) > n^{\log_b a}}$

The Master Theorem (Cont'd)

- Situations that don't look anything like that of the Master Theorem
- T(n) = 2T(n-3) + $\sqrt{n}$
- T(n) = T(n/10) + T(9n/10) + n

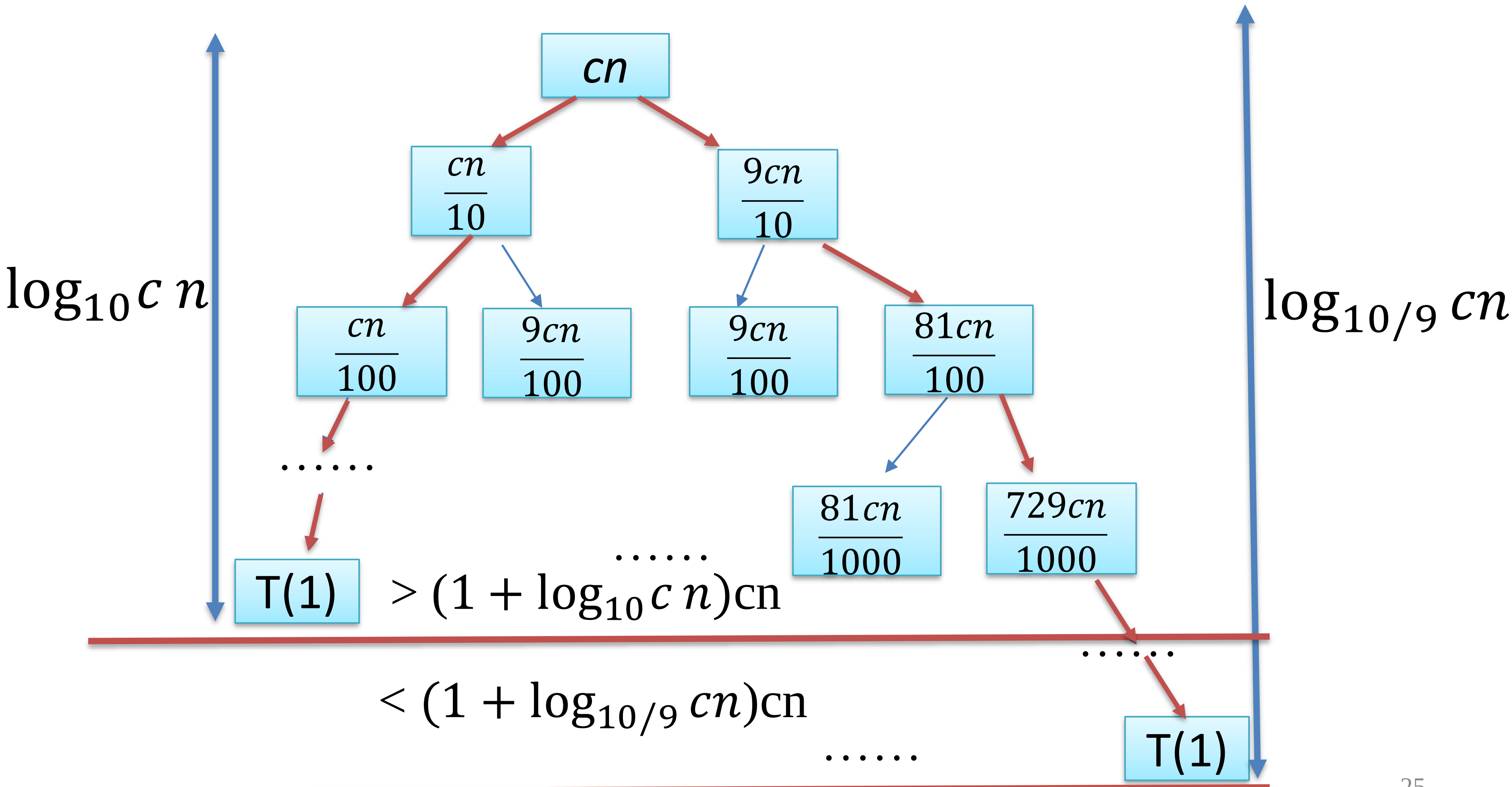
What to do when it doesn't apply

- The recursion-tree method



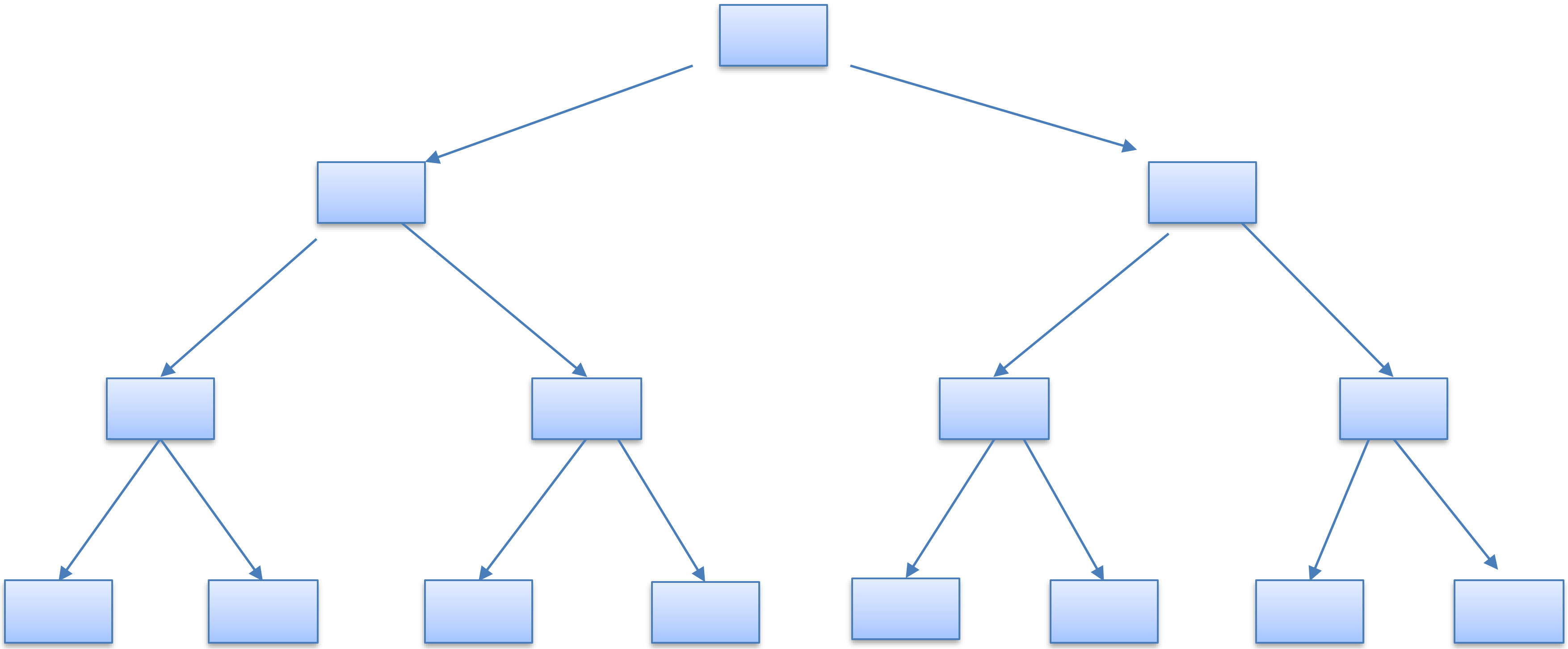
Recursion tree

- T(n) = T(9n/10) + T(n/10) + cn



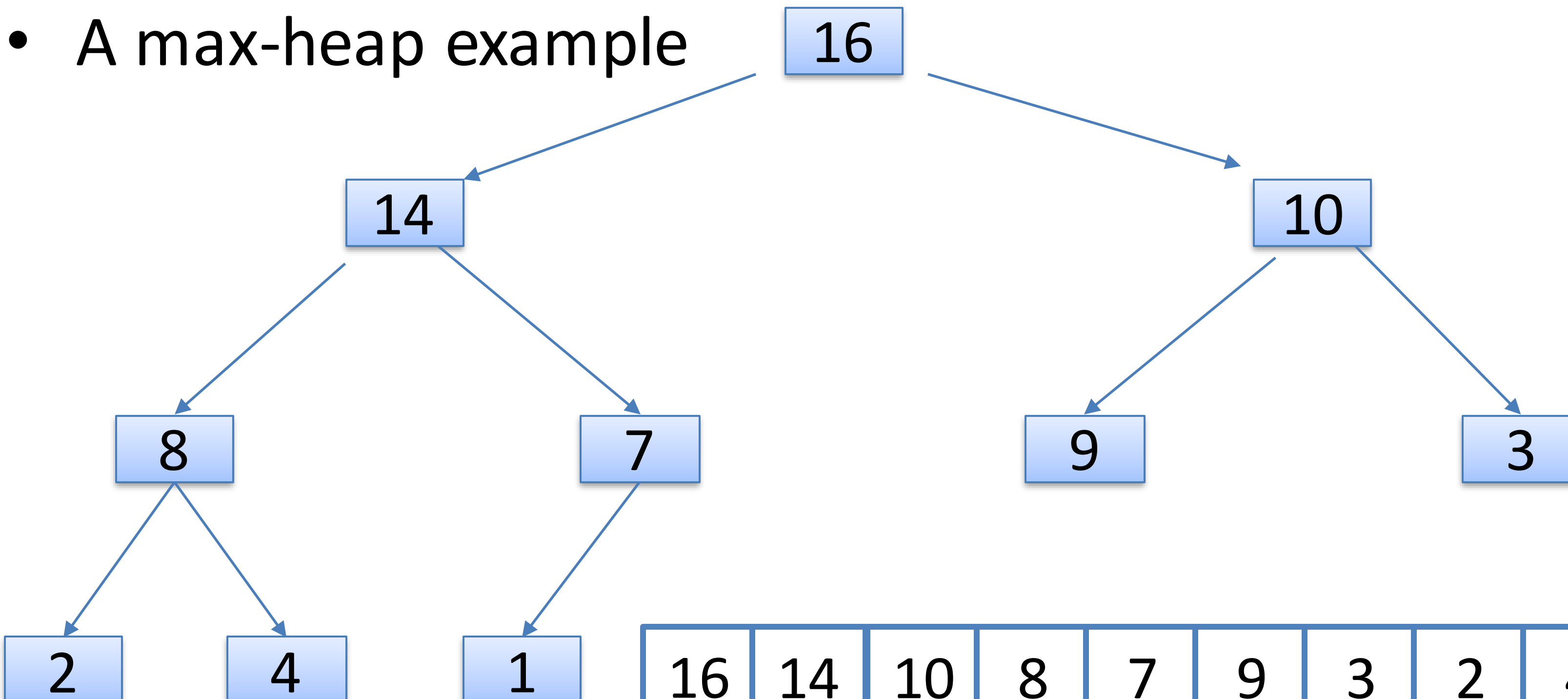
Heaps

- A heap is a complete binary tree or a nearly complete binary tree;



The implementation of heap

- Heaps are usually implemented as arrays (element index starts from 1)
- A max-heap example



16	14	10	8	7	9	3	2	4	1

Referencing heap elements

- So, we have

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```

The property of a heap

- Heaps must satisfy the heap property
- Max-heap:
  - A[parent[i]] $\geq$ A[i] for all nodes $i > 1$
  - In other words, the value of a node is at most the value of its parent
  - The largest element in a max-heap is stored in A[1]

The property of a heap (cont'd)

- Min-heap:
  - A[parent[i]] $\leq$ A[i] for all nodes $i > 1$
  - In other words, the value of a node is at least the value of its parent
  - The smallest element in a min-heap is stored in A[1]

MAX-Heapify () (cont'd)

```
Max_Heapify(A, i)
{
  l = Left(i); r = Right(i);
  if (l <= A.heap_size && A[l] > A[i])
      largest = l;
  else
      largest = i;
  if (r <= A.heap_size && A[r] > A[largest])
      largest = r;
  if (largest != i)
      Swap(A, i, largest);
      Max_Heapify(A, largest);//why this works?
}
```

Build-MAX-Heap()

```
// given an unsorted array A, make A a heap
Build-MAX-Heap(A)
{
  A.heap_size = A.length;
  for (i = ⌊A.length/2⌋ downto 1)
    MAX-Heapify(A, i);
}
```

Heapsort

- Given **`Build-MAX-Heap()`**, an in-place sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard by swapping it with element at A[n]
    - Decrement A.heap_size
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling **`MAX-Heapify()`**
  - Repeat, always swapping A[1] for A[A.heap_size]

Heapsort (cont’d)

```
Heapsort(A)
{
    Build-MAX-Heap(A);
    for (i = A.length downto 2)
    {
        Swap(A[1], A[i]);
        A.heap_size= A.heap_size - 1;
        MAX-Heapify(A, 1);
    }
}
```

Priority Queues

- The heap data structure is incredibly useful for implementing (max-/min-) *priority queues*
  - A data structure for maintaining a set *S* of elements, each with an associated value or key
  - Supports the operations **`Insert()`**, **`Maximum()`**, and **`ExtractMax()`**

Priority Queue Operations

- **Insert(S, x)** inserts the element x into set S
- **Maximum(S)** returns the element of S with the maximum key
- **ExtractMax(S)** removes and returns the element of S with the maximum key
- How could we implement these operations using a heap?

Implementing Priority Queues

```
Heap-Maximum(A)
{
      return A[1];
}
```

Implementing Priority Queues

```
Heap-Extract-Max(A)
{
    if(A.heap_size < 1) { error; }
    max = A[1];
    A[1] = A[A.heap_size];
    A.heap_size = A.heap_size - 1;
    MAX-Heapify(A, 1);
    return max;
}
```

Implementing Priority Queues

```
Max-Heap-Insert(A, key)
{
  A.heap_size = A.heap_size + 1;
  A[A.heap_size]= -∞;
  Heap-INCREASE-KEY(A,A.heap_size,key);
}
//what's running time?
```

Implementing Priority Queues

```
Heap-INCREASE-KEY(A, i, key)

   if key < A[i] {error;}
   A[i]= key;
   while (i>1 and A[PARENT(i)]< A[i])
       exchange(A[i], A[PARENT(i)];
       i= PARENT(i);
} what's running time?
```