Quicksort

- Quick sort
- Correctness of partition
  - loop invariant
- Performance analysis
  - Recurrence relations

Performance

- A triumph of analysis by C.A.R. Hoare

- Worst-case execution time – $\Theta(n^2)$.
- Average-case execution time – $\Theta(n \lg n)$.
  - » How do the above compare with the complexities of other sorting algorithms?

- Empirical and analytical studies show that quicksort can be ***expected*** to be twice as fast as its competitors.

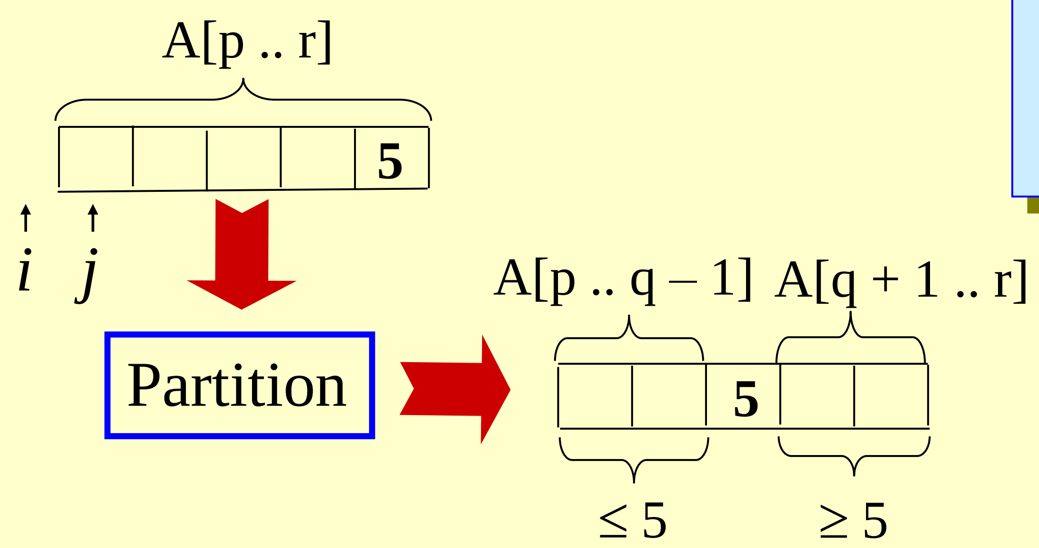
Design

- Follows the **divide-and-conquer** paradigm.
- ***Divide*:** Partition (separate) the array $A[p .. r]$ into two (possibly empty) subarrays $A[p .. q–1]$ and $A[q+1 .. r]$.
  - » Each element in $A[p .. q–1] \leq A[q]$.
  - » $A[q] <$ each element in $A[q+1 .. r]$.
  - » Index q is often referred to as a pivot.
- ***Conquer*:** Sort the two subarrays by recursive calls to quicksort.
- ***Combine*:** The subarrays are sorted in place – no work is needed to combine them.
- How do the divide and combine steps of quicksort compare with those of merge sort?

Pseudocode

```
Quicksort(A, p, r)
    if p < r then
        q := Partition(A, p, r);
        Quicksort(A, p, q – 1);
        Quicksort(A, q + 1, r)
    fi
```

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```



Example

		p									r	
initially:		2	5	8	3	9	4	1	7	10	**6**	
	i	j										
next iteration:		2	5	8	3	9	4	1	7	10	**6**	
		i	j									
next iteration:		2	5	8	3	9	4	1	7	10	**6**	
			i	j								
next iteration:		2	5	8	3	9	4	1	7	10	**6**	
			i		j							
next iteration:		2	5	3	8	9	4	1	7	10	**6**	
				i		j						

note: pivot (x) = 6

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```

Example (Continued)

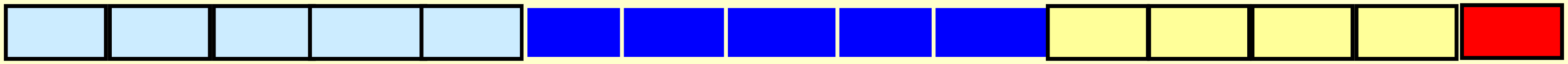
next iteration:	2	5	3	8	9	4	1	7	10	**6**
			i		j					
next iteration:	2	5	3	8	9	4	1	7	10	**6**
			i			j				
next iteration:	2	5	3	4	9	8	1	7	10	**6**
				i			j			
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i			j		
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i				j	
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i					j
after final swap:	2	5	3	4	1	**6**	9	7	10	8
					i					j

Partition(A, p, r)

x, i := A[r], p – 1;
for j := p **to** r – 1 **do**
  if A[j] ≤ x **then**
    i := i + 1;
    A[i] ↔ A[j]
  fi
od;
A[i + 1] ↔ A[r];
return i + 1

Partitioning

- Select the last element $A[r]$ in the subarray $A[p .. r]$ as the *pivot* – the element around which to partition.
- As the procedure executes, the array is partitioned into four (possibly empty) regions.
  1. $A[p .. i]$ — All entries in this region are **≤ *pivot***.
  2. $A[i+1 .. j-1]$ — All entries in this region are > ***pivot***.
  3. $A[j .. r-1]$ — Not known how they compare to *pivot*.
  4. $A[r] =$ *pivot*.
- The above hold before each iteration of the *for* loop, and constitute a *loop invariant*. (4 is not part of the LI - loop invariant.)



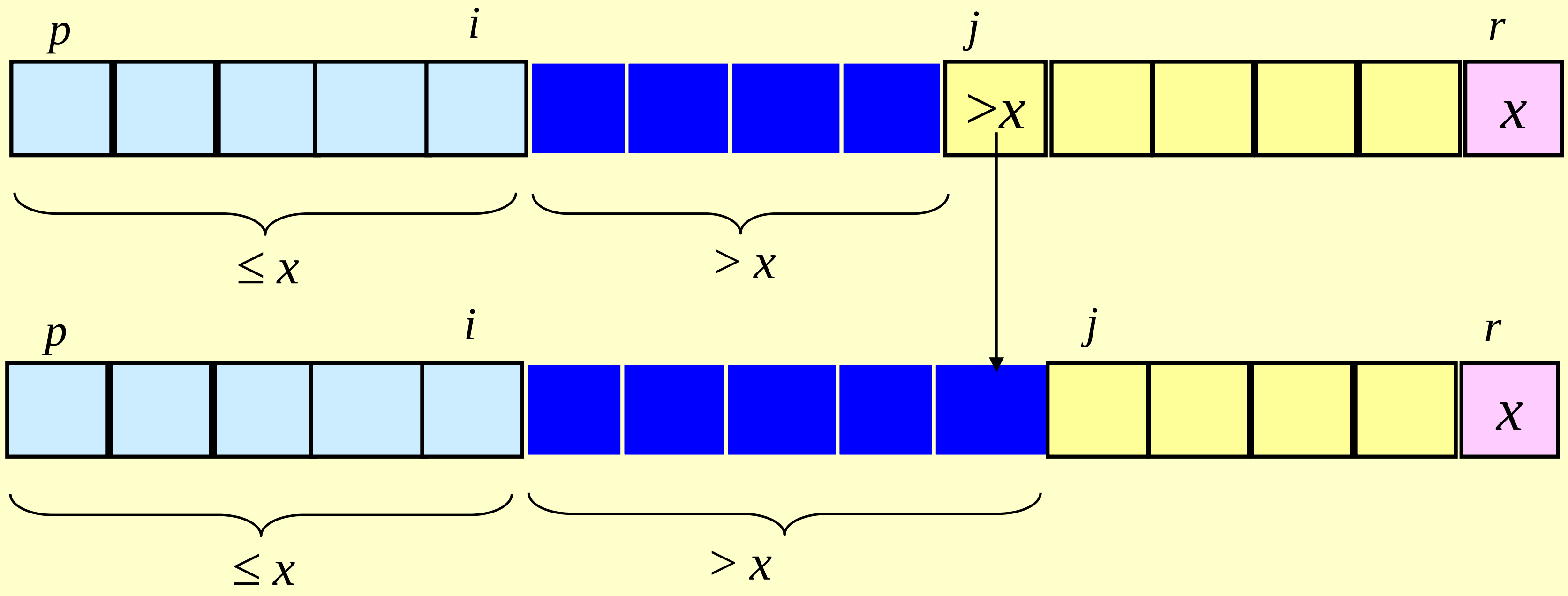
Correctness of Partition

- Use loop invariant.
- **Initialization:**
  - » Before first iteration
    - $A[p..\ i]$ and $A[i + 1\ ..\ j - 1]$ are empty – Conds. 1 and 2 are satisfied (trivially).
    - r is the index of the *pivot* – Cond. 4 is satisfied.
    - Cond. 3 trivially holds.
- **Maintenance:**
  - » **Case 1:** $A[j] > x$
    - Increment j only.
    - LI is maintained.

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```

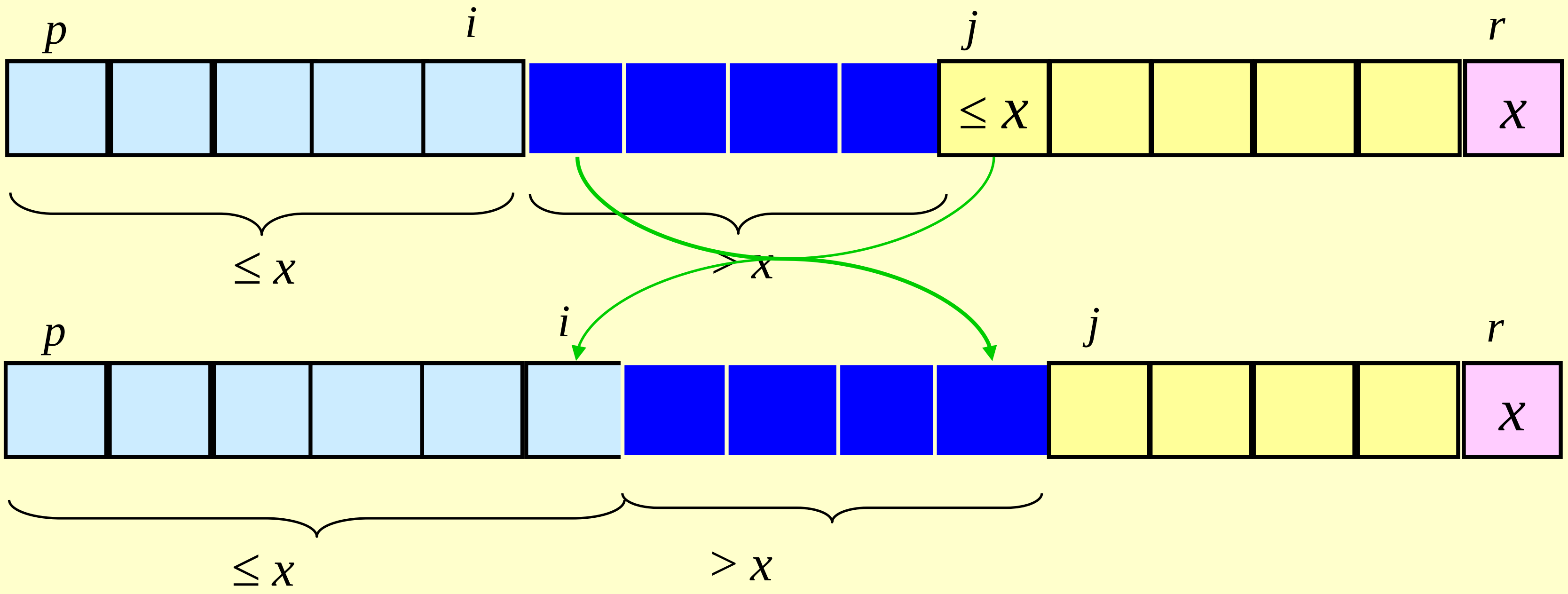
Correctness of Partition

Case 1: $A[j] > x$



Correctness of Partition

- **Case 2:** $A[j] \leq x$
  - » Increment i
  - » Swap $A[i]$ and $A[j]$
    - Condition 1 is maintained.
  - » Increment j
    - Condition 2 is maintained.
  - » $A[r]$ is unaltered.
    - Condition 3 is maintained.



Correctness of Partition

- **Termination:**
  - » When the loop terminates, $j = r$, so all elements in A are partitioned into one of the three cases:
    - $A[p .. i] \leq$ ***pivot***
    - $A[i + 1 .. r - 1] >$ ***pivot***
    - $A[r] =$ ***pivot***
- The last two lines swap $A[i + 1]$ and $A[r]$.
  - » *Pivot* moves from the end of the array to between the two subarrays.
  - » Thus, procedure *partition* correctly performs the divide step.

Complexity of Partition

- PartitionTime(n) is given by the number of iterations in the *for* loop.
- $\Theta(n)$: $n = r - p + 1$.

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```

Algorithm Performance

Running time of quicksort depends on whether the partitioning is balanced or not.

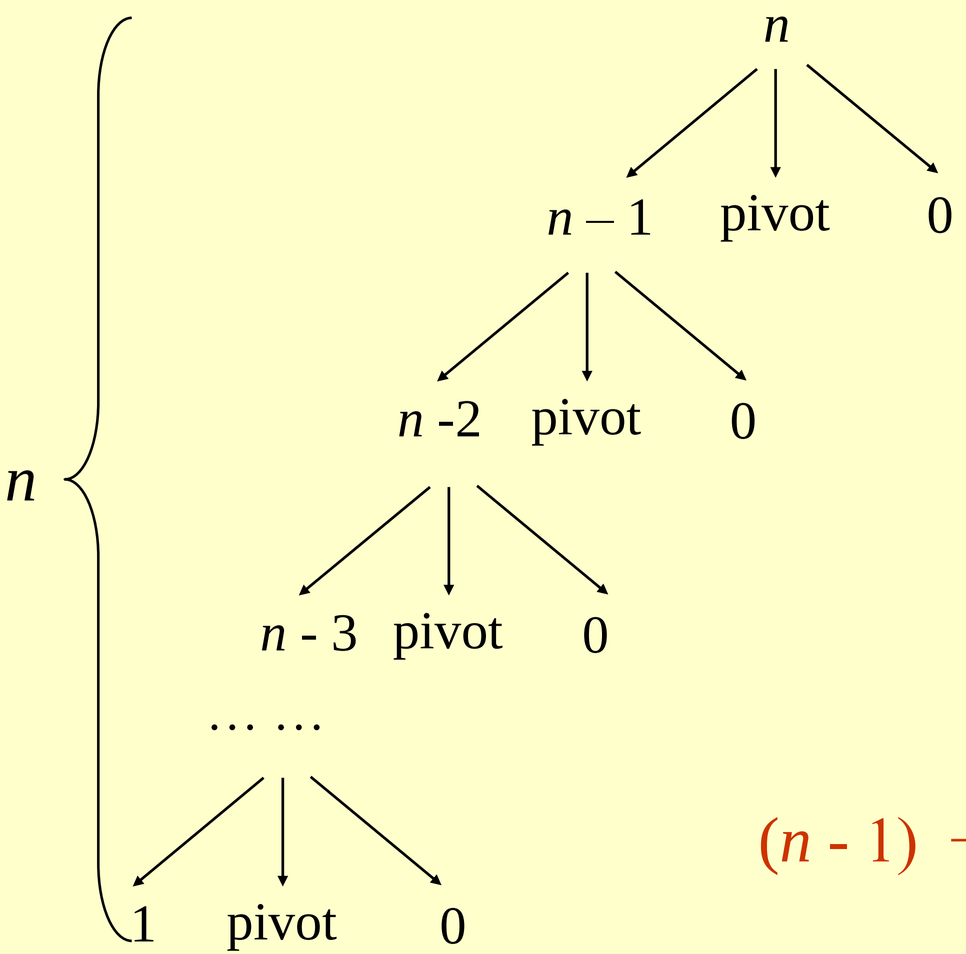
- **Worst-Case Partitioning (Unbalanced Partitions):**
  - » Occurs when every call to partition results in the most unbalanced partition.
  - » Partition is most unbalanced when
    - Subproblem 1 is of size $n - 1$, and subproblem 2 is of size 0 or vice versa.
    - *pivot* $\geq$ every element in $A[p .. r - 1]$ or *pivot* $<$ every element in $A[p .. r - 1]$.
  - » Every call to partition is most unbalanced when
    - Array $A[1 .. n]$ is sorted or reverse sorted!

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

↑ i ↑ j

Worst-case Partition Analysis

Recursion tree for worst-case partition



Running time for worst-case partition at each recursive level:

$T(n) = T(n-1) + T(0)$

$\quad + \text{PartitionTime}(n)$

$\quad = T(n-1) + \Theta(n)$

$\quad = \sum_{k=1 \text{ to } n} \Theta(k)$

$\quad = \Theta(\sum_{k=1 \text{ to } n} k)$

$\quad = \Theta(n^2)$

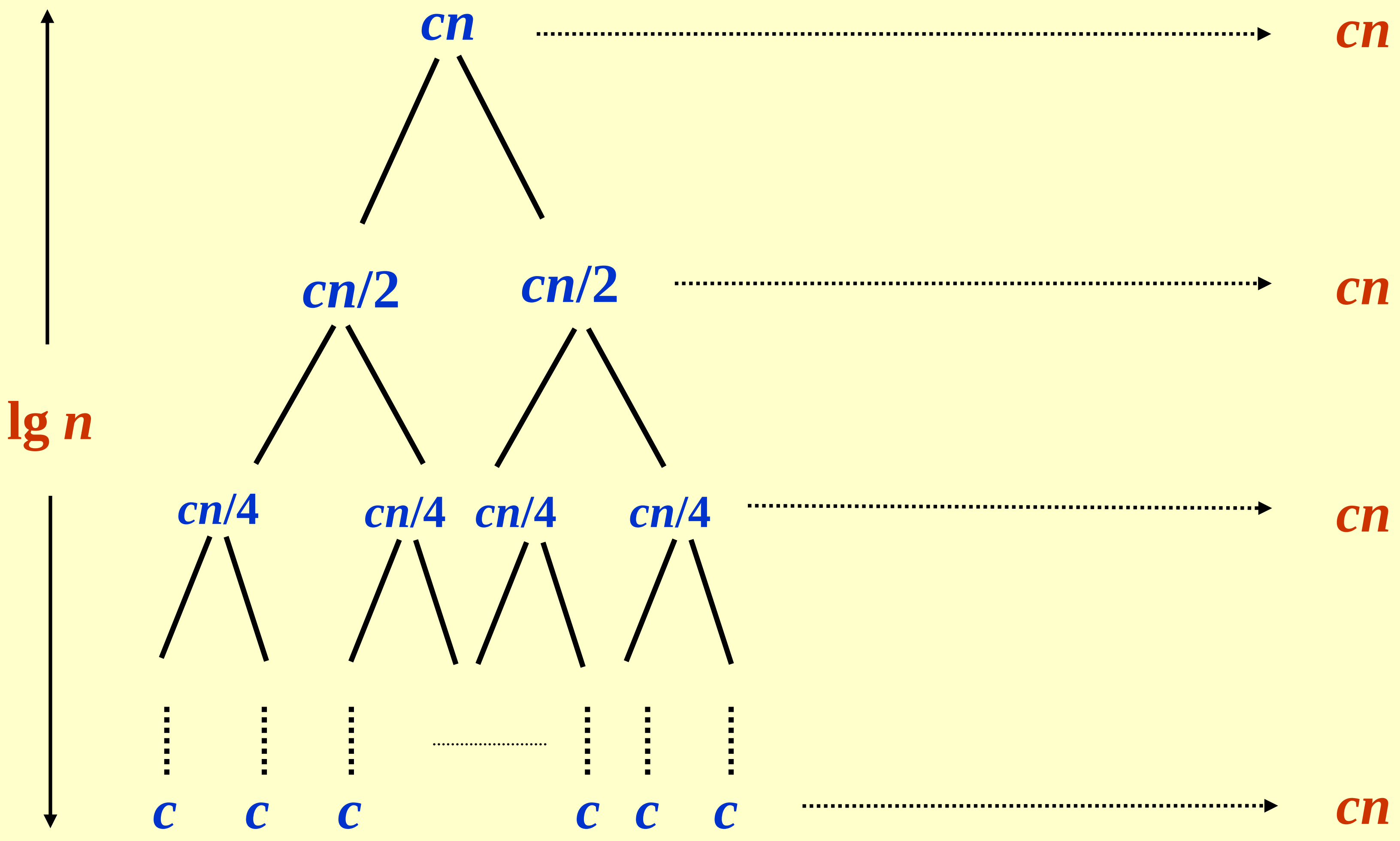
$(n - 1) + \ldots + 1 = n(n-1)/2 = O(n^2)$

Best-case Partitioning

- Size of each subproblem $\le n/2$.
  - » One of the subproblems is of size $\lfloor n/2 \rfloor$
  - » The other is of size $\lceil n/2 \rceil - 1$.
- Recurrence for running time
  - » $T(n) \le 2T(n/2) + \text{PartitionTime}(n)$

    $= 2T(n/2) + \Theta(n)$
- $\boldsymbol{T(n) = \Theta(n \lg n)}$

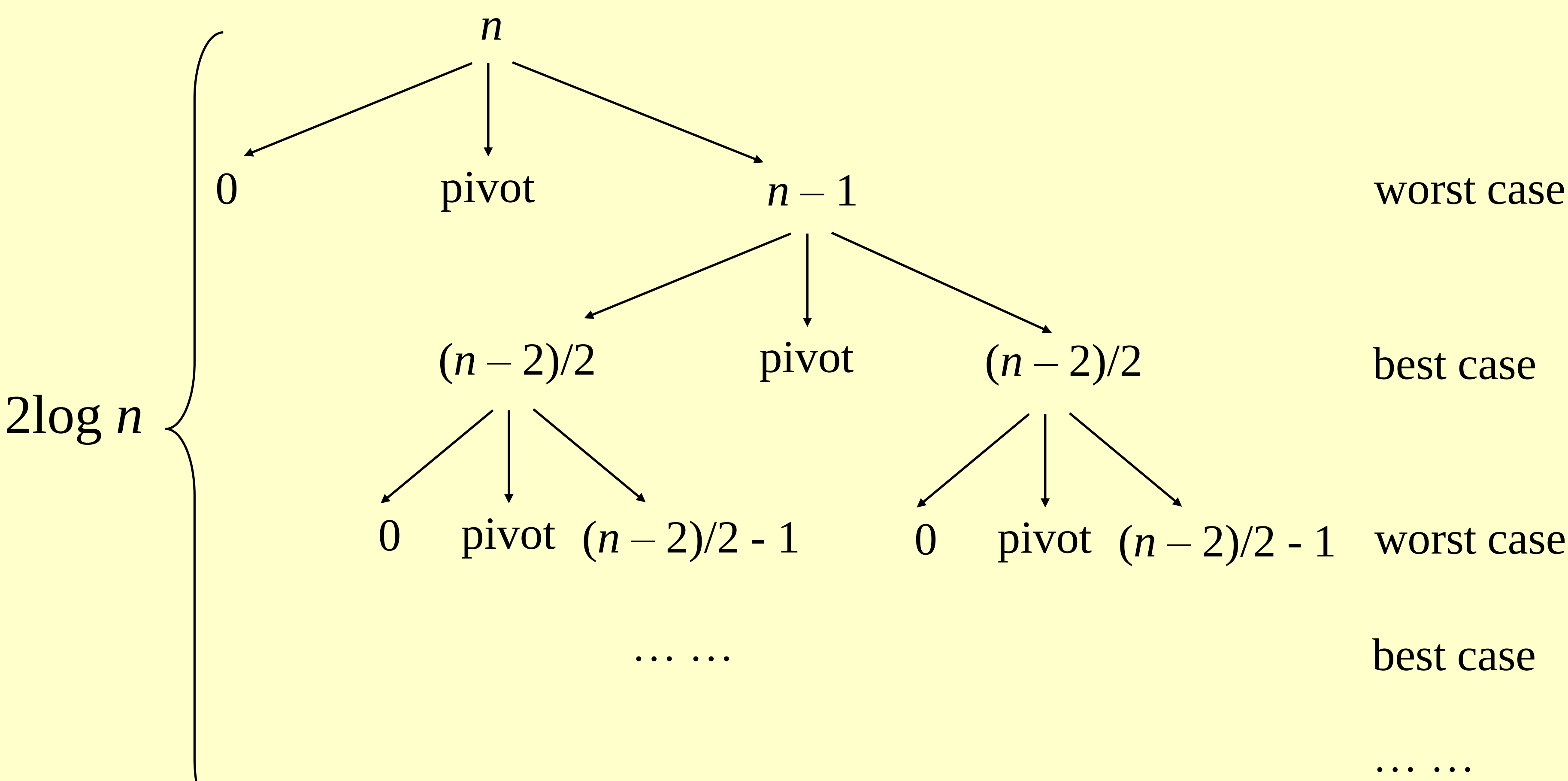
Recursion Tree for Best-case Partition



Total : $O(n \lg n)$

Average-case Partitioning

Average case: Worst cases and best cases interleavingly appear.



Average case time complexity: $\leq 2n \log n$

Recurrences – II

Recurrence Relations

- Equation or an inequality that characterizes a function by its values on smaller inputs.
- **Solution Methods** (Chapter 4)
  - » Substitution Method.
  - » Recursion-tree Method.
  - » Master Method.
- Recurrence relations **arise when we analyze the running time of iterative or recursive algorithms**.
  - » **Ex:** Divide and Conquer.

$$T(n) = \Theta(1) \quad \text{if } n \leq c$$
$$T(n) = a\,T(n/b) + D(n) + C(n) \quad \text{otherwise}$$

Technicalities

- We can (almost always) ignore floors and ceilings.
- Exact vs. Asymptotic functions.
  - » In algorithm analysis, both the recurrence and its solution are expressed using asymptotic notation.
  - » Ex: Recurrence with exact function

$$T(n) = 1 \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + n \quad \text{if } n > 1$$

Solution: $T(n) = n \lg n + n$

- Recurrence with asymptotics (BEWARE!)

$$T(n) = \Theta(1) \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \quad \text{if } n > 1$$

Solution: $T(n) = \Theta(n \lg n)$

- "With asymptotics" means we are being sloppy about the exact base case and non-recursive time – still convert to exact, though!

Substitution Method

- **Guess** the form of the solution, then **use mathematical induction** to show it correct.
  - » Substitute guessed answer for the function when the inductive hypothesis is applied to smaller values – hence, the name.
- Works well when the solution is easy to guess.
- No general way to guess the correct solution.

Example – Exact Function

Recurrence: $T(n) = 1$ if $n = 1$

$T(n) = 2T(n/2) + n$ if $n > 1$

- Guess: $T(n) = n \lg n + n$.
- Induction:
  - **Basis:** $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$.
  - **Hypothesis:** $T(k) = k \lg k + k$ for all $k < n$.
  - **Inductive Step:** $T(n) = 2\,T(n/2) + n$

$$= 2\,((n/2)\lg(n/2) + (n/2)) + n$$

$$= n\,(\lg(n/2)) + 2n$$

$$= n \lg n - n + 2n$$

$$= n \lg n + n$$

Example – With Asymptotics

To Solve: $T(n) = 3T(\lfloor n/3 \rfloor) + n$

- Guess: $T(n) = O(n \lg n)$
- Need to prove: $T(n) \le cn \lg n$, for some $c > 0$.
- Hypothesis: $T(k) \le ck \lg k$, for all $k < n$.
- Calculate:

$$
\begin{aligned}
T(n) &\le 3c \lfloor n/3 \rfloor \lg \lfloor n/3 \rfloor + n \\
&\le c\, n \lg (n/3) + n \\
&= c\, n \lg n - c\, n \lg 3 + n \\
&= c\, n \lg n - n (c \lg 3 - 1) \\
&\le c\, n \lg n
\end{aligned}
$$

(The last step is true for $c \ge 1 / \lg 3$.)

Example – With Asymptotics

To Solve: $T(n) = 3T(\lfloor n/3 \rfloor) + n$

- To show $T(n) = \Theta(n \lg n)$, must show both upper and lower bounds, i.e., $T(n) = O(n \lg n)$ **AND** $T(n) = \Omega(n \lg n)$
- (Can you find the mistake in this derivation?)
- Show: $T(n) = \Omega(n \lg n)$
- Calculate:

$$\begin{aligned} T(n) &\geq 3c \lfloor n/3 \rfloor \lg \lfloor n/3 \rfloor + n \\ &\geq c\, n \lg (n/3) + n \\ &= c\, n \lg n - c\, n \lg 3 \;+ n \\ &= c\, n \lg n - n\, (c \lg 3 - 1) \\ &\geq c\, n \lg n \end{aligned}$$

(The last step is true for $c \leq 1 / \lg 3$.)

Example – With Asymptotics

If $T(n) = 3T(\lfloor n/3 \rfloor) + O(n)$, as opposed to $T(n) = 3T(\lfloor n/3 \rfloor) + n$, then rewrite $T(n) \le 3T(\lfloor n/3 \rfloor) + cn$, $c > 0$.

- To show $T(n) = O(n \lg n)$, use second constant $\boldsymbol{d}$, different from $\boldsymbol{c}$.
- Calculate:

$$
\begin{aligned}
T(n) &\le 3d \lfloor n/3 \rfloor \lg \lfloor n/3 \rfloor + c\,n \\
&\le d\,n \lg (n/3) + cn \\
&= d\,n \lg n - d\,n \lg 3 + cn \\
&= d\,n \lg n - n\,(d \lg 3 - \mathrm{c}) \\
&\le d\,n \lg n
\end{aligned}
$$

(The last step is true for $d \ge c / \lg 3$.)

It is <u>OK</u> for d to depend on c.

Making a Good Guess

- If a recurrence is similar to one seen before, then guess a similar solution.
  - » $T(n) = 3T(\lfloor n/3 \rfloor + 5) + n$ (Similar to $T(n) = 3T(\lfloor n/3 \rfloor) + n$)
    - When n is large, the difference between $n/3$ and $(n/3 + 5)$ is insignificant.
    - Hence, can guess $O(n \lg n)$.
- Method 2: Prove loose upper and lower bounds on the recurrence and then reduce the range of uncertainty.
  - » E.g., start with $T(n) = \Omega(n)$ & $T(n) = O(n^2)$.
  - » Then lower the upper bound and raise the lower bound.

Subtleties

- When the math doesn't quite work out in the induction, strengthen the guess by subtracting a lower-order term. Example:
  - » Initial guess: $T(n) = O(n)$ for $T(n) = 3T(\lfloor n/3 \rfloor) + 4$
  - » Results in: $T(n) \le 3c\lfloor n/3 \rfloor + 4 = c\,n + 4$
  - » Strengthen the guess to: $T(n) \le c\,n - b$, where $b \ge 0$.
    - What does it mean to strengthen?
    - Though counterintuitive, it works. Why?

$T(n) \le 3(c\lfloor n/3 \rfloor - b) + 4 \le c\,n - 3b + 4 = c\,n - b - (2b - 4)$

Therefore, $T(n) \le c\,n - b$, if $2b - 4 \ge 0$ or if $b \ge 2$.

(Don't forget to check the base case: here $c > b + 1$.)

Changing Variables

- Use algebraic manipulation to turn an unknown recurrence into one similar to what you have seen before.
  - » Example: $T(n) = 2T(n^{1/2}) + \lg n$
  - » Rename $\boldsymbol{m = \lg n}$ and we have

    $T(2^m) = 2T(2^{m/2}) + m$
  - » Set $S(m) = T(2^m)$ and we have

    $S(m) = 2S(m/2) + m \Rightarrow S(m) = O(m \lg m)$
  - » Changing back from $S(m)$ to $T(n)$, we have

    $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$

Avoiding Pitfalls

- Be careful not to misuse asymptotic notation. For example:
  - » We can falsely prove $T(n) = O(n)$ by guessing $T(n) \le cn$ for $T(n) = 2T(\lfloor n/2 \rfloor) + n$

$$T(n) \le 2c\lfloor n/2 \rfloor + n$$
$$\le c\,n + n$$
$$= O(n) \quad \Leftarrow \text{Wrong!}$$

  - » We are supposed to prove that $T(n) \le c\,n$ for all $n > N$, according to the definition of $O(n)$.
- Remember: prove the *exact form* of inductive hypothesis.

Exercises

- Solution of $T(n) = T(\lceil n/2 \rceil) + n$ is $O(n)$
- Solution of $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ is $O(n \lg n)$
- Solve $T(n) = 2T(n/2) + 1$
- Solve $T(n) = 2T(n^{1/2}) + 1$ by making a change of variables. Don't worry about whether values are integral.