Insertion Sort: Runtime

```
for  i = 2 to n do                                          line 1
     key = A[i]                                             line 2
     //Insert A[i] into the sorted subarray A[1, i − 1]
     j = i − 1                                              line 3
     while j > 0 and A[j] > key do                          line 4
          A[j + 1] = A[j]                                   line 5
          j = j − 1                                         line 6
     end while
     A[j + 1] = key                                         line 7
end for
```

For $2 \le i \le n$, let t_i = #times line 4 is executed. Then

$$T(n) = n + 3(n-1) + \sum_{i=2}^{n} t_i + 2\sum_{i=2}^{n}(t_i - 1) = 3\sum_{i=2}^{n} t_i + 2n - 1$$

- Which (size n) input yields the smallest (best-case) running time?
- Which (size n) input yields the largest (worst-case) running time?

Worst case analysis

Worst-case running time: largest possible running time of the algorithm over all inputs of a given size n.

Why worst-case analysis?

- It gives well-defined computable bounds.
- Average-case analysis can be tricky: how do we generate a "random" instance?

The worst-case running time of insertion-sort is quadratic.

So… is insertion-sort efficient?

Efficiency of Insertion sort

Compare to brute force solution:

- At each step, generate a new permutation of the n integers.
- If sorted, stop and output the permutation.

Worst-case analysis: generate n! permutations. Is brute force solution *efficient*?

- Efficiency relates to the performance of the algorithm as n grows.
- Stirling's approximation formula: $n! \approx (n/e)^n$
  - For n = 10, generate $3.67^{10} \geq 2^{10}$ permutations.
  - For n = 50, generate $18.3^{50} \geq 2^{200}$ permutations.
  - For n = 100, generate $36.7^{100} \geq 2^{700}$ permutations!

⇒ Brute force solution is not efficient.

Efficiency of Algorithms

Definition (attempt 1) An algorithm is efficient if it achieves better worst-case performance than brute-force search.

Caveat: fails to discuss the scaling properties of the algorithm; if the input size grows by a **constant factor**, we would like the running time T(n) of the algorithm to increase by a constant factor as well.

Polynomial running times: on input of size n, T(n) is at most $c \cdot n^d$ for c,d > 0 constants.

- Polynomial running times scale well!
- The smaller the exponent of the polynomial the better.

Efficiency of Algorithms

Definition An algorithm is efficient if it has a polynomial running time.

Caveat

- What about huge constants in front of the leading term or large exponents?

However

- Small degree polynomial running times exist for most problems that can be solved in polynomial time.
- Conversely, problems for which no polynomial-time algorithm is known tend to be very hard in practice.
- So we can distinguish between easy and hard problems.

Efficiency of Insertion sort

So by our definition Insertion sort *is* efficient. …are we done with sorting?

Can we do better?

What exactly is better? n^2 vs. $(3/2)n^2 + (7/2)n - 4$?

To discuss this, we need a coarser classification of running times of algorithms; exact characterizations

- are too detailed;
- do not reveal similarities between running times in an immediate way as n grows large;
- are often **meaningless**: pseudocode steps will **expand** by a constant factor that depends on the hardware.

Asymptotic Notation

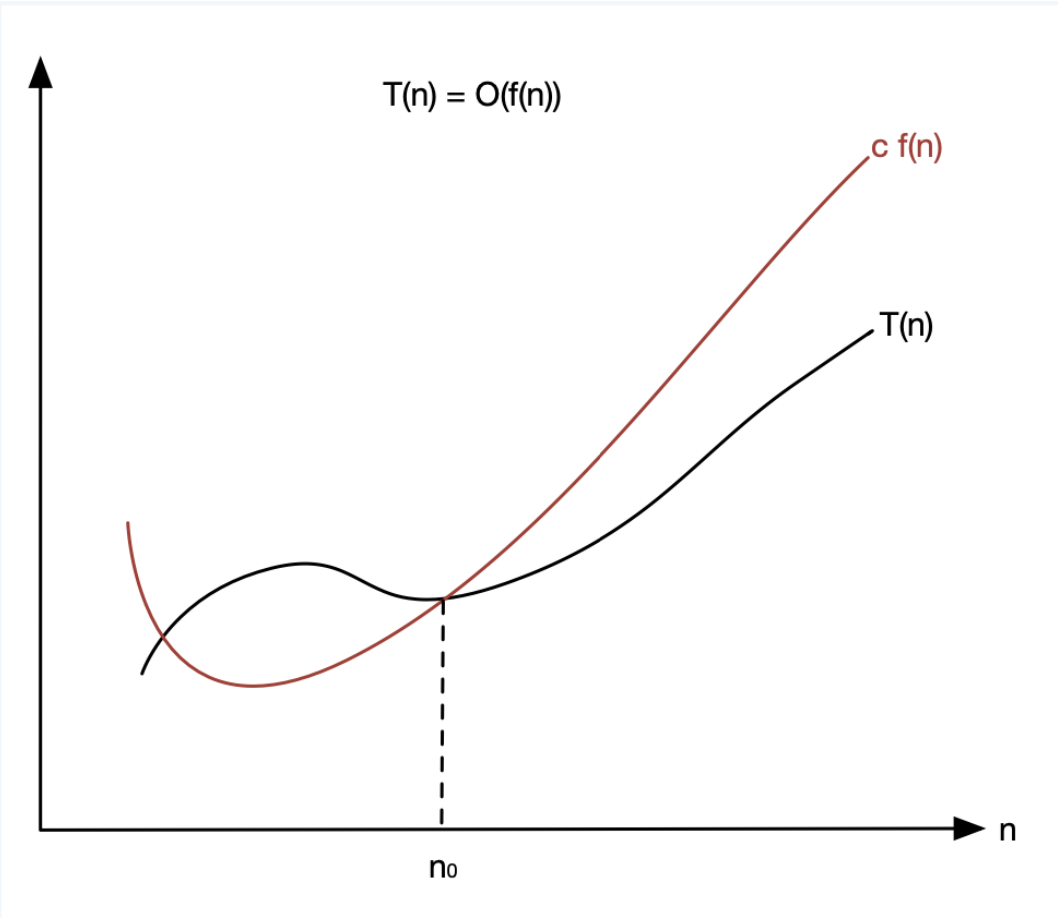
A framework that will allow us to compare the rate of growth of different running times as the input size n grows.

- We will express the running time as a function of the number of primitive steps, which is a function of the size of the input n.

- To compare functions expressing running times, we will ignore their low-order terms and focus solely on the highest-order term.

Asymptotic upper bounds: Big-O notation

Definition (Big-O) We say that T(n) = O(f(n)) if there exist constants c > 0 and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$



Asymptotic upper bounds: Big-O notation

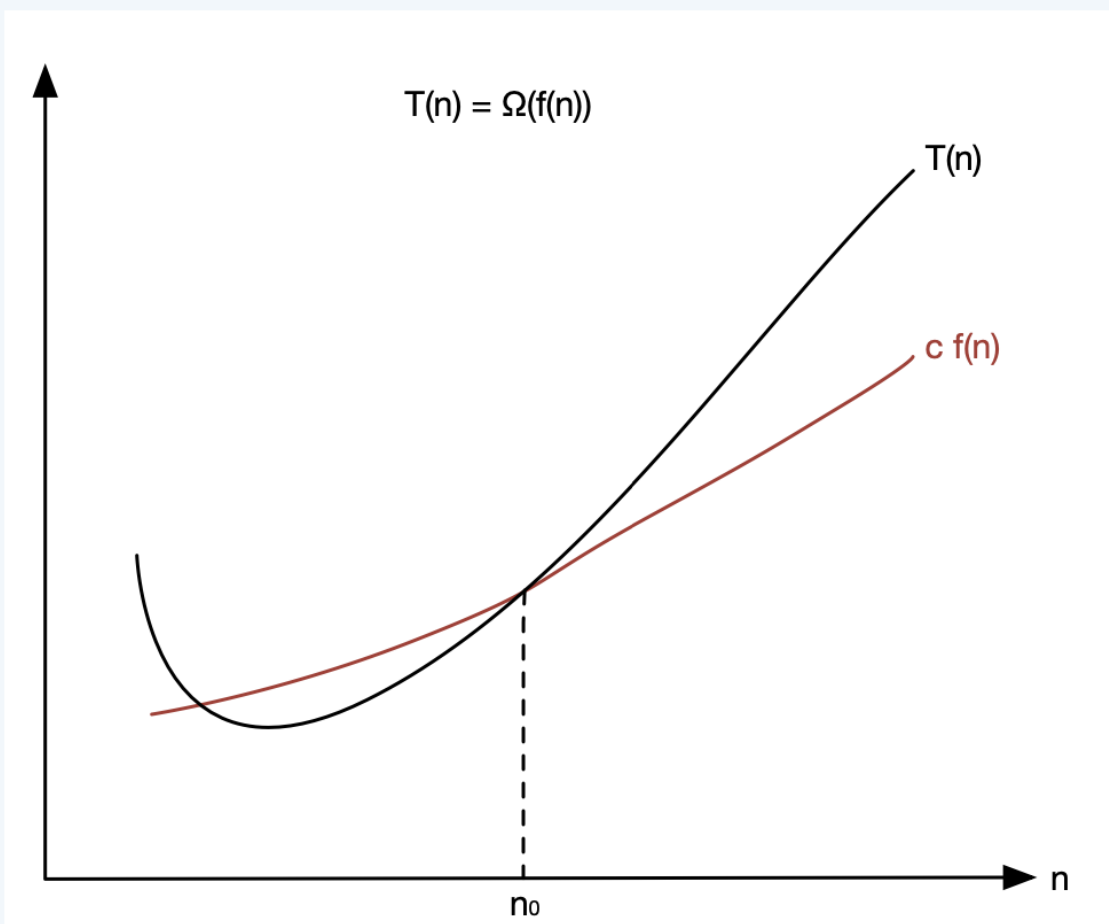
Definition (Big-O) We say that $T(n) = O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$

Examples: Show that $T(n) = O(f(n))$ when

- $T(n) = an^2 + b$, $a,b>0$ constants and $f(n) = n^2$.
- $T(n) = an^2 + b$ and $f(n) = n^3$.

Asymptotic lower bounds: Big-Ω notation

Definition (Big-Ω) We say that $T(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \geq c \cdot f(n)$



Asymptotic lower bounds: Big-Ω notation

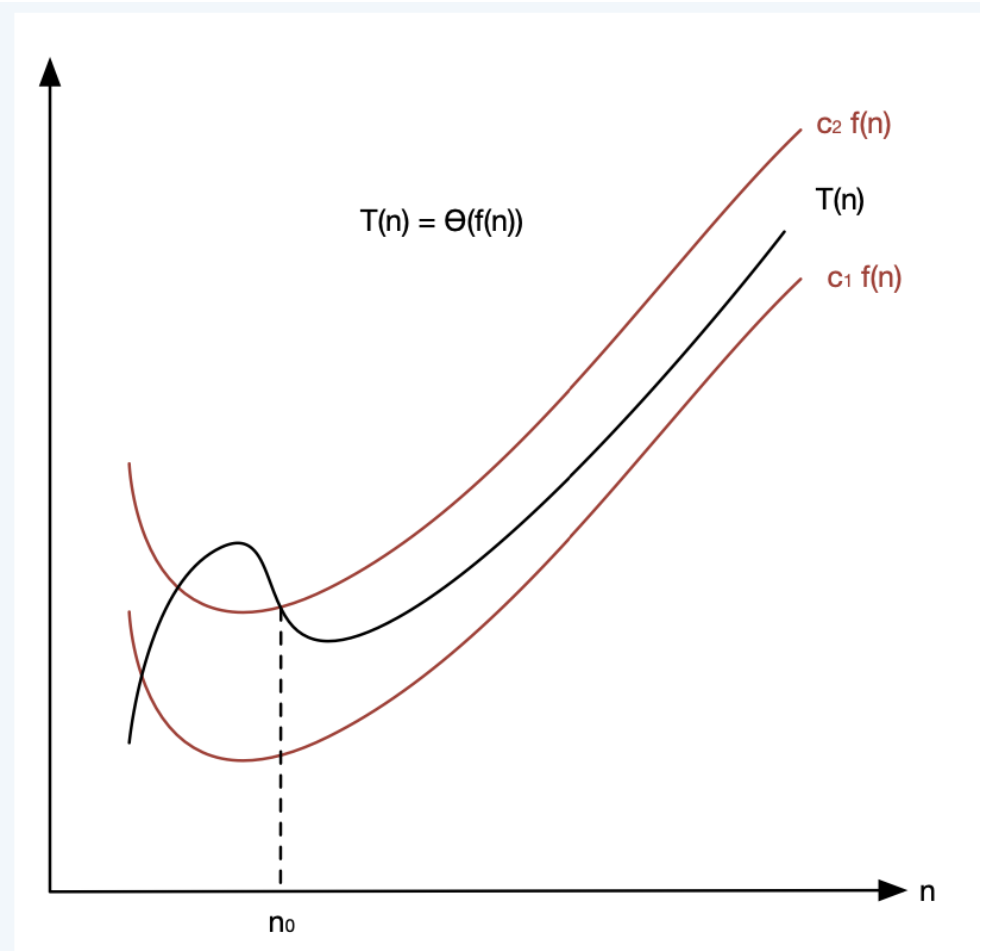
Definition (Big-Ω) We say that $T(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \geq c \cdot f(n)$

Examples: Show that $T(n) = O(f(n))$ when

- $T(n) = an^2 + b$, $a,b>0$ constants and $f(n) = n^2$.
- $T(n) = an^2 + b$ and $f(n) = n$.

Asymptotic tight bounds: Big-Θ notation

Definition (Big-Θ) We say that $T(n) = \Theta(f(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$



Asymptotic tight bounds: Big-Θ notation

Definition (Big-Θ) We say that $T(n) = \Theta(f(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$

Equivalent definition

$T(n) = \Theta(f(n))$ iff $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

Notational convention: log n stands for $\log_2 n$

Examples: Show that $T(n) = \Theta(f(n))$ when

- $T(n) = an^2 + b$, $a,b>0$ constants and $f(n) = n^2$
- $T(n) = n \log n + n$ and $f(n) = n \log n$

Asymptotic upper bounds: little-*o*

Definition (little-*o*) We say that T(n) = *o*(f(n)) if for any constant c > 0, there exists $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) < c \cdot f(n)$

- Intuitively, T(n) becomes insignificant relative to f(n) as $n \to \infty$.
- Proof by showing that $\lim_{n\to\infty} \frac{T(n)}{f(n)} = 0$ (if the limit exists).

Examples: Show that T(n) = *o*(f(n)) when

- $T(n) = an^2 + b$, a,b>0 constants and $f(n) = n^3$.
- $T(n) = n \log n$ and $f(n) = n^2$.

Asymptotic lower bounds: little-ω

Definition (little-ω) We say that T(n) = ω(f(n)) if for any constant c > 0, there exists $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have T(n) > c·f(n)

- Intuitively, T(n) becomes arbitrarily large relative to f(n) as $n \to \infty$.
- Proof by showing that $\lim_{n\to\infty} \frac{T(n)}{f(n)} = \infty$ (if the limit exists).

Examples: Show that T(n) = ω(f(n)) when

- $T(n) = 2^n$, a,b>0 constants and $f(n) = n^3$.
- $T(n) = n^2$ and $f(n) = n \log n$.

Basic rules for omitting low order terms

- Ignore multiplicative factors: e.g., $10n^3$ becomes n^3
- n^a dominates n^b if $a > b$: e.g., n^2 dominates n
- Exponentials dominate polynomials: e.g., 2^n dominates n^4
- Polynomials dominate logarithms: e.g., n dominates $\log^3 n$

$\Rightarrow$ For large enough n,

$$\log n < n < n \log n < n^2 < n^{200} < 2^n < 3^n < n^n$$

Properties of asymptotic growth rates

Transitivity

- If $f = O(g)$ and $g = O(h)$, then $f = O(h)$
- If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$
- If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$

Sums of up to a constant number of functions

- If $f = O(h)$ and $g = O(h)$, then $f+g = O(h)$
- Let k be a fixed constant, and let $f_1, f_2, \ldots, f_k, h$ be functions such that for all i, $f_i = O(h)$. Then $f_1 + f_2 + \ldots + f_k = O(h)$

Transpose symmetry

- $f = O(g)$ if and only if $g = \Omega(f)$
- $f = o(g)$ if and only if $g = \omega(f)$

The Divide-and-Conquer Principle

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively.
- **Combine** the solutions to the subproblems to get the solution to the overall problem.

Can sometimes help in designing more efficient algorithms (eg. Sorting!)

Divide-and-Conquer applied to Sorting

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.

  Divide the input array into two lists of equal size.

- **Conquer** the subproblems by solving them recursively.

  Sort each list recursively. (Stop when lists have size 2.)

- **Combine** the solutions to the subproblems to get the solution to the overall problem.

  Merge the two sorted lists and output the sorted array.

aka mergesort!

mergesort: pseudocode

mergesort (A,left,right)

if right == left **then** return **endif**
mid = left + ⌊(right − left)/2⌋

mergesort (A,left,mid)

mergesort (A, mid + 1, right)

merge (A, left, right, mid)

Remarks

- mergesort is a recursive procedure (*why?*)
- Initial call: **mergesort**(A,1,n)
- Subroutine **merge** merges two sorted lists of sizes ⌊n/2⌋, ⌈n/2⌉ into one sorted list of size n. *How can we accomplish this?*

merge: intuition

Intuition: To merge two sorted lists of size n/2 repeatedly

- compare the two items in the front of the two lists;
- extract the smaller item and append it to the output;
- update the front of the list from which the item was extracted.

Example: n = 8, L = (1,3,5,7), R = (2,6,8,10)

merge: pseudocode

merge (A, left, right, mid)

L = A[left, mid]

R = A[mid + 1, right]

Maintain two pointers p_L,p_R, initialized to point to the first elements of L, R, respectively

while both lists are nonempty **do**

Let x, y be the elements pointed to by p_L,p_R

Compare x, y and append the smaller to the output

Advance the pointer in the list with the smaller of x, y

end while

Append the remainder of the non-empty list to the output.

Remark: the output is stored directly in A[left,right], thus the subarray A[left,right] is sorted after merge(A,left,right,mid).

merge: analysis

Correctness: by induction on the size of the two lists

(recommended exercise)

Running time:

- L, R have n/2 elements each
- How many iterations before all elements from both lists have
- been appended to the output? At most n − 1.
- How much work within each iteration? Constant.

⇒ merge takes O(n) time to merge L, R (why?)

Space: extra Θ(n) space to store L, R (the output of merge is stored directly in A).

mergesort: correctness

For simplicity, assume $n = 2^k$ for integer $k \geq 0$.

We will use induction on k.

- **Base** case: For k = 0, the input consists of 1 item; mergesort returns the item.
- **Induction Hypothesis**: For $k \geq 0$, assume that mergesort correctly sorts any list of size 2^k.
- **Induction Step**: We will show that mergesort correctly sorts any list A of size 2^{k+1}.

  From the pseudocode of mergesort, we have:

  - Line 3: mid takes the value 2^k
  - Line 4: **mergesort**(A, 1, 2^k) correctly sorts the leftmost half of the input, by the induction hypothesis.
  - Line 5: **mergesort**(A, $2^k + 1$, 2^{k+1}) correctly sorts the rightmost half of the input, by the induction hypothesis.
  - Line 6: **merge** correctly merges its two sorted input lists into one sorted output of size $2^k + 2^k$

$\Rightarrow$ **mergesort** correctly sorts any input of size 2^{k+1}.

mergesort: running time

The running time of **mergesort** satisfies:

$$T(n) = 2T(n/2) + cn, \text{ for } n \geq 2, \text{ constant } c > 0$$

$$T(1) = c$$

This structure is typical of recurrence relations

- an **inequality** or **equation** bounds T(n) in terms of an expression involving T (m) for m < n
- a base case generally says that T (n) is constant for small constant n

Remarks

- We ignore floor and ceiling notations.
- A recurrence does **not** provide an asymptotic bound for T(n): to this end, we must **solve** the recurrence.

Solving recurrences: recursion trees

The technique consists of three steps

1. Analyze the first few levels of the tree of recursive calls
2. Identify a pattern
3. Sum the work spent over all levels of recursion

Example: give an asymptotic bound for the recurrence describing the running time of **mergesort**

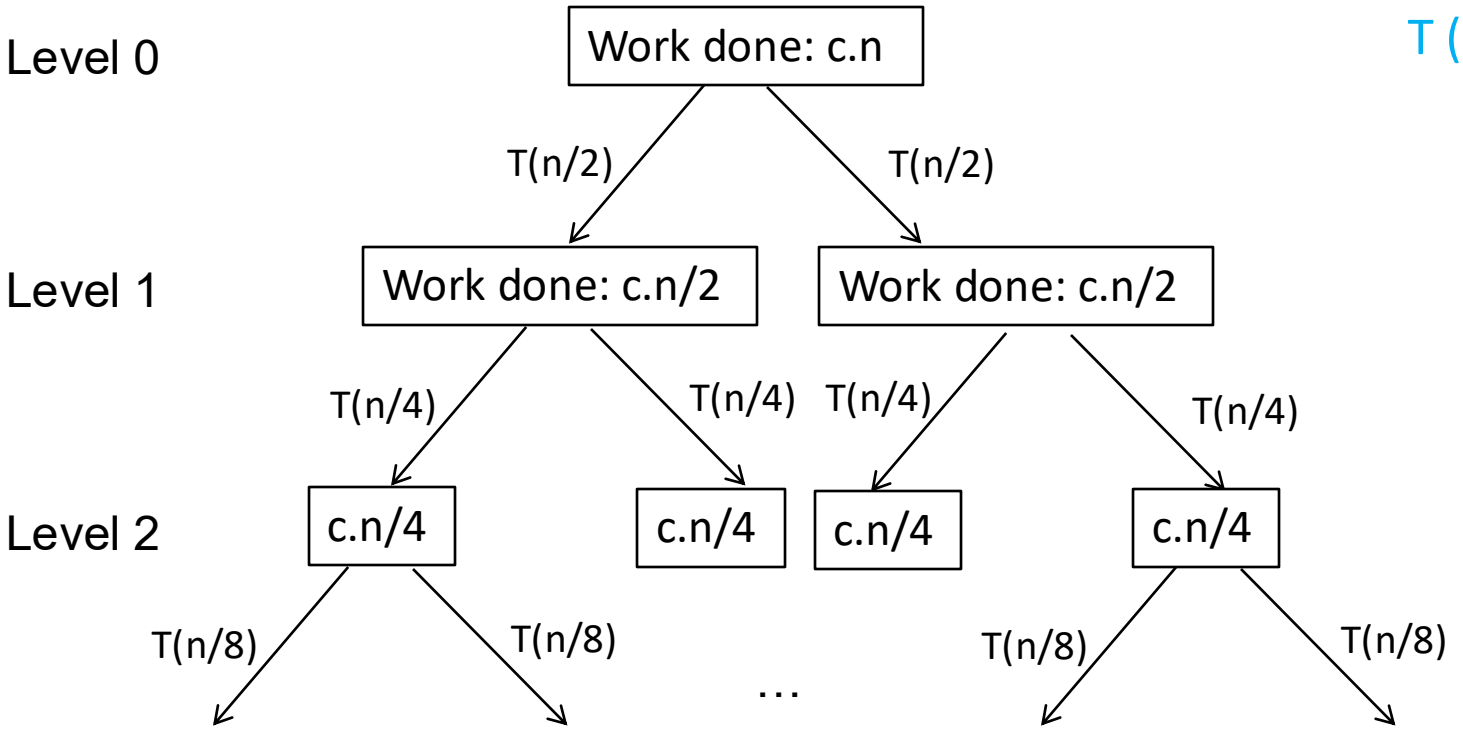
$T(n) = 2T(n/2) + cn$, for $n \geq 2$, constant $c > 0$

$T(1) = c$

Solving recurrences: recursion trees

$T(n) = 2T(n/2) + cn$

$T(1) = c$



Observe:

- At each level i, total work done is: $2^i \cdot c\,(n/2^i) = cn$
- Let d be the max number of levels, then $2^d \leq n$. Equivalently $d \leq \log n$

Therefore: total runtime for mergesort: $cn \cdot d = cn \log n = O(n \log n)$

A general recurrence and solution

The running times of many recursive algorithms can be expressed by the following recurrence

$$T(n) = a\,T(n/b) + cn^k \text{ , for } a, c > 0,\ b > 1, k \geq 0$$

What is the recursion tree for this recurrence?

- a is the branching factor
- b is the factor by which the size of each subproblem shrinks

  ⇒ at level i, there are a^i subproblems, each of size n/b^i

  ⇒ each subproblem at level i requires $c(n/b^i)^k$ work
- the height of the tree is $\log_b n$ levels

⇒ Total work done: $\sum_{i=0}^{\log_b n} a^i c\left(\frac{n}{b^i}\right)^k \ \sum_{i=0}^{\log_b n} a^i c\left(\frac{n}{b^i}\right)^k = cn^k \sum_{i=0}^{\log_b n}\left(\frac{a}{b^k}\right)^i$

Solving recurrences

Theorem (Master theorem):

If $T(n) = a\,T(\lceil n/b \rceil) + O(n^k)$ for some constants $a > 0$, $b > 1$, $k \geq 0$, then

$$T(n) = \begin{cases} O(n^{\log_b a}) & , \text{if } a > b^k \\ O(n^k \log n) & , \text{if } a = b^k \\ O(n^k) & , \text{if } a < b^k \end{cases}$$

Example: running time of **mergesort**

- $T(n) = 2T(n/2) + cn$

  $a=2, b=2, k=1, b^k=2=a \Rightarrow T(n) = O(n \log n)$

Solving recurrences

Theorem (Master theorem):

If $T(n) = a\,T(\lceil n/b \rceil) + O(n^k)$ for some constants $a > 0$, $b > 1$, $k \geq 0$, then

$$T(n) = \begin{cases} O(n^{\log_b a}) & , \text{if } a > b^k \\ O(n^k \log n) & , \text{if } a = b^k \\ O(n^k) & , \text{if } a < b^k \end{cases}$$

Important:

- Not all recurrences can be solved by above Theorem.
- Examples:
  - $T(n) = 2T(n-1) + 1$, where $T(1) = 2$
  - $T(n) = 2T^2(n-1)$, where $T(1) = 4$

Divide-and-Conquer Example 2: Searching

Searching in a sorted array

Input: sorted list A of n integers, and an integer x

Output:

index j such that $1 \leq j \leq n$ and A[j]=x; or

no if x is not in A

Example: A = {0,2,3,5,6,7,9,11,13}, n = 9, x = 7

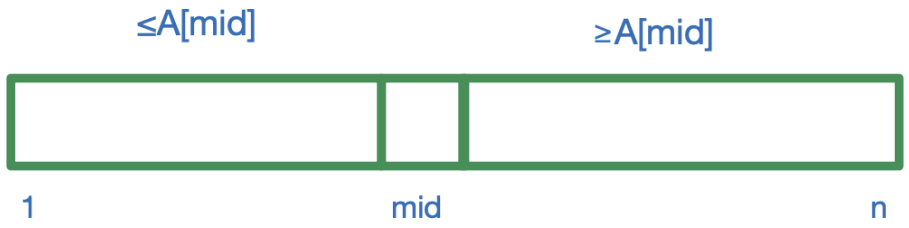
Idea: use the fact that the array is sorted and probe specific entries in the array.

Binary Search

First, probe the middle entry. Let mid = ⌈n/2⌉.

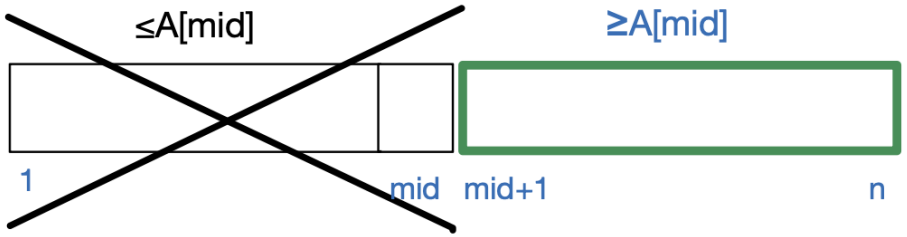
- If x == A[mid], return mid
- If x < A[mid] then look for x in A[1,mid−1];
- Else if x > A[mid] look for x in A[mid+1,n].

Initially, the entire array is "active", that is, x might be anywhere in the array.



Suppose x > A[mid].

Then the active area of the array, where x might be, is to the right of mid.



Binary Search: pseudocode

```
binarysearch(A, left, right)
    mid = left + [(right − left)/2]
    if x == A[mid] then
        return mid
    else if right == left then
        return no
    else if x > A[mid] then
        left = mid + 1
    else right = mid − 1
    endif
    binarysearch(A, left, right)
```

Initial call: **binarysearch**(A, 1, n)

Binary Search: analysis

Correctness: (try it at home)

Space: O(1)

Runtime:

At each step there is a region of A where x could be and we shrink the size of this region by a factor of 2. Hence the recurrence for the running time is

$$T(n) \leq T(n/2) + O(1)$$

By Master theorem (a=1, b=2, k=0), we have $T(n) = O(\log n)$