5 Divide & conquer

[BB chapter 7] or [Par chapter 2.1]

5.1 Merge Sort

[BB chapter 7.4]

$\boxed{\textbf{Problem:}}$ Given an array of integers $A[1 \dots n]$. Sort the array.

Idea:

- Divide the n element sequence into two subsequences with $n/2$ elements each.
- Sort the two sequences recursively.
- Merge the two sequences to produce the answer

```
// sort sequence A[l..r]
function merge_sort(l,r)
  // base case - 1 element is always sorted
  if (l=r) then return;
  m=(l+r)/2;
  // we need to sort sequences l..m, m+1..r
  merge_sort(l,m);
  merge_sort(m+1,r);
  // and finally merge two sorted sequences
  merge(l,m,r);

//merge two sorted sequences l..m, m+1..r
function merge(l,m,r)
  copy A[l..m] to L; L[m-l+2]:=infinity;
  copy A[m+1..r] to R; R[r-m+1]:=infinity;

  i:=1; j:=1; k:=l;
  while (L[i]<infinity or R[j]<infinity) do
    if L[i]<=R[j] then
      A[k]:=L[i];
      i:=i+1; k:=k+1;
    else
      A[k]:=R[j];
      j:=j+1; k:=k+1;
```

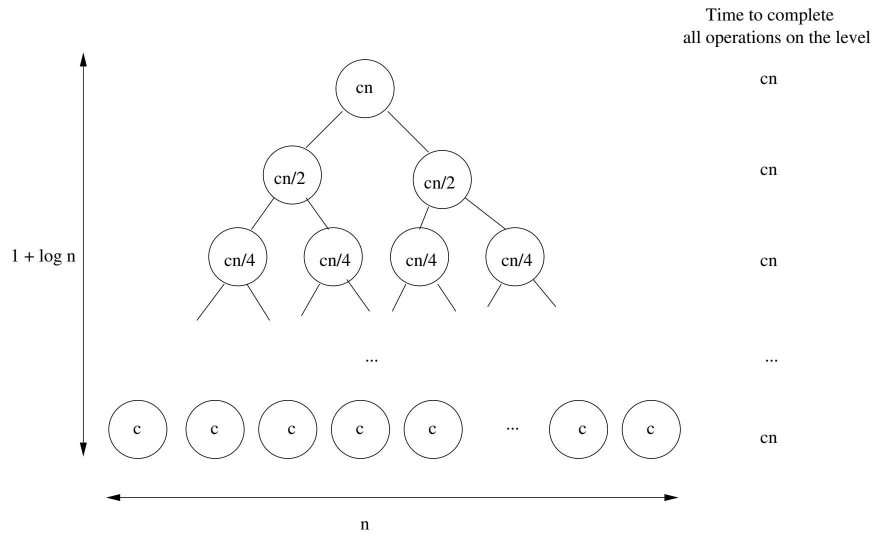
Time: Denote $T(n)$ the time which is needed to sort n element array by `merge_sort`. Then we can say:

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn \tag{1}$$

$$T(1) \leq c \tag{2}$$

Omit some details and obtain a guess. Assume for now that $n = 2^k$; thus we can write:

$$T(n) \le 2T(n/2) + cn$$



Based on this recursion tree we can guess that $T(n) \le c'n(1 + \log n)$.

Now prove the guess more precisely.

Lemma 1. *For all $n \ge 1$, $T(n) \le c'n(1 + \log n)$, for some constant c' (to be determined later).*

Proof. By induction on value of n.

Base case: Assume $n = 1$. Then from (2) we have $T(1) \le c \le c'n(1 + \log n)$ (as far as $c' \ge c$)

Induction step: Assume that $n > 1$ and $T(n') \le c'n'(1 + \log n')$ for all $n' < n$. Then we have:

$$\begin{array}{rll}
T(n) & \le T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + cn & \text{from (1)} \\
& \le c' \lfloor \frac{n}{2} \rfloor (1 + \log \lfloor \frac{n}{2} \rfloor) + c' \lceil \frac{n}{2} \rceil (1 + \log \lceil \frac{n}{2} \rceil) + cn & \text{from IH} \\
& = c' \lfloor \frac{n}{2} \rfloor \log \lfloor \frac{n}{2} \rfloor + c' \lceil \frac{n}{2} \rceil \log \lceil \frac{n}{2} \rceil + c'n + cn & \text{because } \lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil = n \\
& \le c' \lfloor \frac{n}{2} \rfloor (\log n - 1) + c' \lceil \frac{n}{2} \rceil \log n + c'n + cn & \text{because } \log \lfloor \frac{n}{2} \rfloor \le \log n/2 = \log n - 1 \\
& & \text{and } \log \lceil \frac{n}{2} \rceil \le \log n \\
& \le c'n \log n - c' \lfloor \frac{n}{2} \rfloor + c'n + cn & \text{because } \lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil = n \\
& \le c'n \log n - c'(\frac{n}{2} - \frac{1}{2}) + c'n + cn & \text{because } \lfloor \frac{n}{2} \rfloor \ge \frac{n}{2} - \frac{1}{2} \\
& \le c'n \log n + \frac{1}{2}c'n + cn + \frac{1}{2}c' &
\end{array}$$

The only thing left to prove is that

$$\frac{1}{2}c'n + cn + \frac{1}{2}c' \le c'n.$$

This is clearly true if $c' \ge 4c$. □

$\boxed{\textbf{Floors and ceilings:}}$ Part of our difficulties were caused by the fact that we had ceilings and floors in our formulas. Without them, the proof would have been much easier.

We will be often "sloppy" in the next few sections:

- We will often assume that our values of n are "nice" and whenever we want to split the dataset it can be split exactly (in particular, if we want to split the dataset in two equally sized parts, we will assume that the dataset is of even size)
- This assumption is often made in the initial stages of the development of the divide and conquer algorithms. However, the details of how to handle cases where the dataset does not split exactly need to be worked out at the end (but, in interest of time, we will often not do that in class). In most cases, these details are straightforward and the running time analysis will work out with floors and ceilings as well (with slightly more complicated proofs).
- If you want to follow the details on floors and ceilings, have a look at [CLRS2, 4.4.2].
- In your assignment/exam questions you may do the same assumptions unless told otherwise. For example, in case of analysis of the merge sort you would use a simpler recurrence $T(n) = 2T(n/2) + cn$.

5.2 Divide & conquer – summary

The technique used in merge sort is called **divide and conquer**.

1. **Divide.** Divide the problem into several smaller instances of the same problem.
   e.g.: *In the merge sort we split the array to be sorted into two arrays of half size each.*
2. **Conquer.** Solve all the subproblems by recursively calling function solving the problem. If subproblems are small enough solve them in a straightforward manner.
   e.g.: *In the merge sort we call the same function recursively for smaller arrays.*
3. **Combine.** Combine the solutions of the smaller problems into a solution of the big problem.
   e.g.: *In the merge sort we need to merge the two sorted sequences to obtain the sorted answer.*

Note: We need to carefully consider the size of the subproblems, the number of subproblems and the time needed to combine the subproblems.

- If there are too many subproblems or they are too large, then too much time is spent on the recursive calls resulting in an inefficient algorithm.
- If time needed to combine subproblems is large, then we spend too much time combining the subproblems, again resulting in an inefficient algorithm.
- Subproblems should be of similar size. Otherwise the large subproblems will dominate the computation.

Examples we have already seen:

- **Quick sort.** You have seen the quick sort which runs in $O(n \log n)$ average case time and $O(n^2)$ worst case time. We will see later a quick sort which runs in $O(n \log n)$ worst case time – and we will see why it is not practical.
- **Bentley's problem - Solution 3.**
- **Binary search.** [BB chapter 7.3] Find given number x in a sorted array A. Short description:
  - Look in the middle element m of the array

  – If $x = m$ we are done, otherwise if $x < m$ look recursively in the left subarray, or if $x > m$ look recursively in the right subarray.

This is, indeed, an example of the divide and conquer method – we decompose our problem into a **single** subproblem of half size.

5.3 Multiplying large numbers

[BB, Section 7.1]

For small numbers that can be represented in 32 bits we consider a multiplication be an elementary operation. What about large numbers?

$\boxed{\textbf{Problem:}}$ Given two arrays $X[1..n]$, $Y[1..n]$ representing two numbers by their decimal digits, i.e.

$$x = \sum_{i=1}^{n} X[i] \cdot 10^{i-1}, \quad y = \sum_{i=1}^{n} Y[i] \cdot 10^{i-1}.$$

Compute number $z = xy$.

Note: An actual implementation would probably use word-size chunks instead of decimal digits. We use decimal digits for better illustration.

Primary school multiplication algorithm:

```
       9 8 1
x    1 2 3 4
-------------
     3 9 2 4
   2 9 4 3
 1 9 6 2
 9 8 1
-------------
1 2 1 0 5 5 4
```

Time: $\Theta(n^2)$

Can we do better? Use divide and conquer.

- Split both numbers in two halves. Let $k = n/2$.
  $x = a \cdot 10^k + b$
  $y = c \cdot 10^k + d$
- Note: $xy = (a \cdot 10^k + b)(c \cdot 10^k + d) = ac \cdot 10^{2k} + (ad + bc) \cdot 10^k + bd$

  Thus we can break the whole operation to:

  – 4 multiplications of size $n/2$ (ac, ad, bc, bd),

  – 3 additions of size $2n$
- Use recursion to compute the 4 multiplications and then perform additions to obtain the result. Additions can be done in linear time in length of the numbers (another primary school algorithm).

- **Time:** $T(n) = 4T(n/2) + \Theta(n)$

  How many leaves we have in the recursion tree? Depth is $\log_2 n$. On each level each of the subproblems is split into 4 subproblems. So we have $4^{\log_2 n} = n^2$ leaves. Each of the leaves take $\Omega(1)$ time thus the time needed is $\Omega(n^2)$ (and that was not counting the work done on the other levels).

  This is not better than the primary school algorithm :-(

What are the expensive operations? We are spliting our problem into too many subproblems. Can we reduce the number of multiplications?

- Yes we can!

$$\begin{aligned} xy &= (a \cdot 10^k + b)(c \cdot 10^k + d) \\ &= ac \cdot 10^{2k} + (ad + bc) \cdot 10^k + bd \\ &= ac \cdot 10^{2k} + ((a+b)(c+d) - ac - bd) \cdot 10^k + bd \end{aligned}$$

  This is only:

  - 3 multiplications (ac, bd, $(a+b)(c+d)$)
  - 6 additions

- **Time:** $T(n) = 3T(n/2) + \Theta(n)$ – this is better, but we do not know how to analyze this. **We need more techniques for analyzing recurrences.**

  (The result is $\Theta(n^{1.5849\cdots})$ – we will prove that later.)

Notes on implementation: This algorithm is not straightforward to implement. Issues:

- How to deal with cases where split is not exact?
- How to deal with cases where the numbers do not have the same length?
- How to minimize overhead and extensive copying of data?
- When to decide NOT to recurse and rather use the primary school algorithm (because overhead would make the algorithm slower)?

Historical notes:

- Multiplication of large numbers is done in practice by other techniques ("Fast Fourier Transform") in $\Theta(n \log n)$ time. (see [CLRS2, Chapter 30] if interested)
- Related problem: multiplication of two $n \times n$ matrices (recall your algebra course to understand what does this mean).
- Trivial solution of the matrix multiplication takes $\Theta(n^3)$ time.
- **Strassen's algorithm** (1969) improves that to $\Theta(n^{2.81\cdots})$ using similar technique as we used for long number multiplication (reducing number of multiplications of smaller submatrices from 8 to 7) – see [CLRS2, Chapter 28.2]
- More efficient algorithms exist for matrix multiplication – the best has running time $O(n^{2.376})$ (1990)

5.4 Methods for solving recurrences

[BB, Chapter 4.7] or [CLRS2, Chapter 4]

We have three methods for solving recurrences:

- Substitution method
- Recursion trees
- The master theorem

5.4.1 Substitution method

Idea: Assume we have a guess for a closed form of running time. We can prove that the guess is correct by induction on n.

Summary:

1. Formulate the guess in exact form
   e.g.: *$T(n) \le cg(n)$ instead of $T(n) = O(g(n))$; the form should also cover the base case*
   Do not fix constant c – it will be determined during the proof.
2. Prove the guess by induction on n, using recurrence in the induction step.

Advantages:

- Easy to do, once we have a proper guess.
- Floor and ceiling issues are already included in the proof.

Disadvantages:

- We need a good guess first.
- We **have to** work with floors and ceilings (because induction is on integers).

Note: The same method can be used to prove lower bounds as well.

Example 1: We have already seen a typical example in time analysis of the merge sort.

Example 2: Consider the same recurrence: $T(n) \le T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + cn$, $T(1) \le c$. What about guess $T(n) = O(n)$?

Lemma 2. $T(n) \le c'n$

Proof. By induction on n.

Base case: OK as far as $c' \ge c$

Induction step:

$$T(n) \le T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + cn \le c' \left\lfloor \frac{n}{2} \right\rfloor + c' \left\lceil \frac{n}{2} \right\rceil + cn = c'n + cn = O(n)$$

□

This is wrong! Why? We did not prove the exact form of the recurrence but we have used it as an induction hypothesis.

Example 3: Consider recurrence $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + c$, $T(1) = c$. We guess that the solution is $T(n) = O(n)$ and thus we are trying to prove:

Lemma 3. $T(n) \le c'n$

Induction step:

$$T(n) \leq c' \left\lfloor \frac{n}{2} \right\rfloor + c' \left\lceil \frac{n}{2} \right\rceil + c = c'n + c$$

...does not work! But we are only off by constant!

Idea: Substract lower order term.

Lemma 4. $T(n) \leq c'n - b$

Induction step:

$$T(n) \leq c' \left\lfloor \frac{n}{2} \right\rfloor - b + c' \left\lceil \frac{n}{2} \right\rceil - b + c = c'n - 2b + c$$

...works! (as far as $c < b$)

5.4.2 Recursion trees

We have seen this method in time analysis of the merge sort as well.

Summary:

- Draw a tree corresponding to recursive calls; each subproblem has a single node in the tree and we label it with amount of time which is spent working on this subproblem (not counting recursive calls – this work is contained in its children).
- Clearly, sum of all nodes of such tree is the running time of the algorithm.
- We can sum the nodes level by level and then sum costs over all levels.

Advantages:

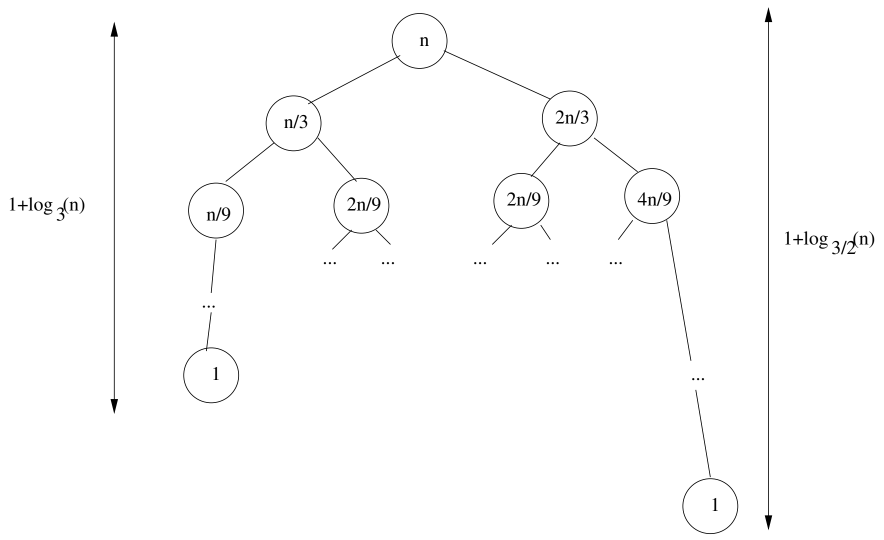
- Recursion trees are easy to build and not too hard to analyze (though summation can be nasty)

Disadvantages:

- It is very hard to deal with floors and ceilings; we are usually "sloppy" and ignore them (but then this does not qualify as a rigorous proof).
- Often there are ... hiding complex claims which should be proven by induction.

This is a very good method for obtaining guesses which can be later proven by the substitution method.

Example: $T(n) = T(n/3) + T(2n/3) + n$, $T(1) = 1$



- Branches end on different levels. Therefore it is not easy to compute the bound directly.
- **Observation 1:** First branch ends on the level $1 + \log_3 n$ and work on each of the levels above sums to **exactly** n. Therefore we have $\Omega(n \log n)$ bound.
- **Observation 2:** The longest branch is $1 + \log_{3/2} n$ long. Sum on each of the level is **at most n**. Therefore we have $O(n \log n)$ bound.
- So the overall bound is $\Theta(n \log n)$.

5.4.3 The master theorem

Theorem 1 (Master theorem)**.** *Let $T(n) = aT(n/b) + f(n)$, $T(1) = \Theta(1)$. Let $k = \log_b a$. Then:*

1. *If $\boxed{f(n) \in O(n^{k-\varepsilon})}$ for some $\varepsilon > 0$ then $\boxed{T(n) \in \Theta(n^k)}$.*
2. *If $\boxed{f(n) \in \Theta(n^k)}$ then $\boxed{T(n) \in \Theta(f(n) \log n)}$.*
3. *If $\boxed{f(n) \in \Omega(n^{k+\varepsilon})}$ for some $\varepsilon > 0$, and regularity condition holds, then $\boxed{T(n) \in \Theta(f(n))}$.*

Regularity condition: *There exists $c < 1$ such that for all sufficiently large n, $af(n/b) \leq cf(n)$.*

Note: The master theorem holds also for reasonable ceiling/floor settings – see [CLRS2 chapter 4.4.2] for details.

Examples of use:

- **Merge sort:** $T(n) = \underbrace{2}_{a} T(n/\underbrace{2}_{b}) + \underbrace{\Theta(n)}_{f(n)}$

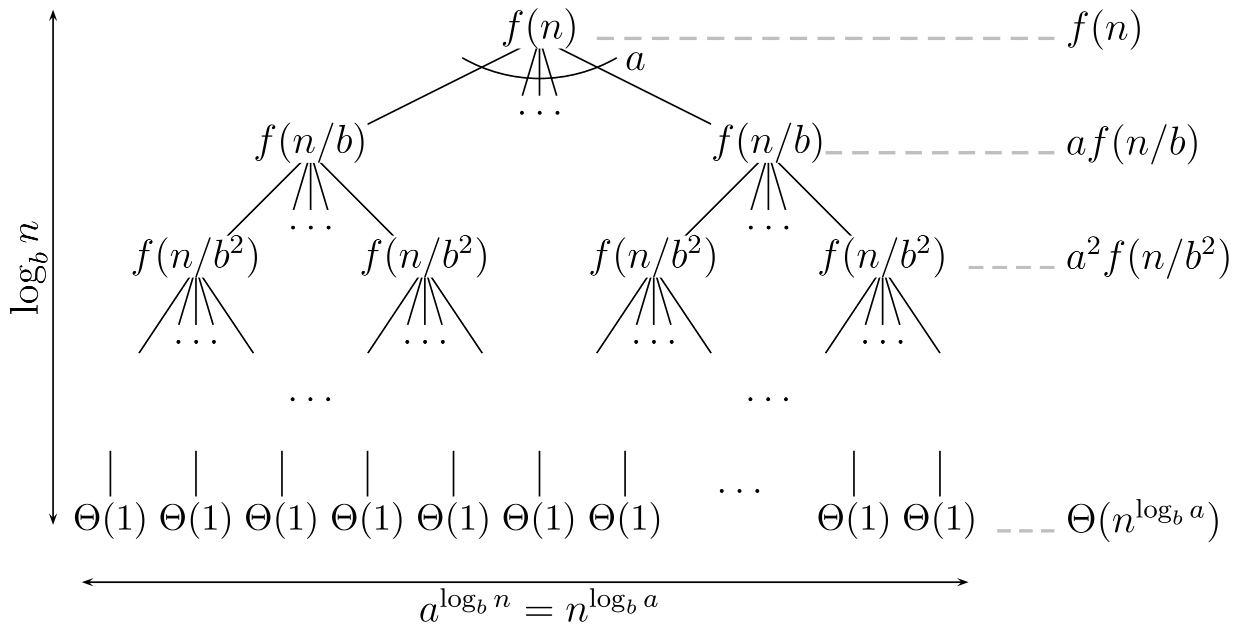
  $k = \log_2 2 = 1,\ f(n) \in \Theta(n) \Rightarrow T(n) \in \Theta(n \log n)$
- **Multiplication of long numbers:** $T(n) = \underbrace{3}_{a} T(n/\underbrace{2}_{b}) + \underbrace{\Theta(n)}_{f(n)}$

  $k = \log_2 3 = 1.5849\ldots,\ f(n) \in O(n^{1.5849\ldots-\varepsilon}) \Rightarrow T(n) \in \Theta(n^{1.5849\ldots})$

- **There are cases where master theorem cannot be used!** $T(n) = 2T(n/2) + n \log n$
  $k = \log_2 2 = 1$, $n \log n \notin \Theta(n)$, $n \log n \notin \Omega(n^{1+\varepsilon})$, $n \log n \notin O(n^{1-\varepsilon})$:-(

5.4.4 Proof sketch of the master theorem

[CLRS2 chapter 4.4]



From the Figure we can see:

$$T(n) = \underbrace{\Theta(n^k)}_{\text{LEAVES}} + \underbrace{\sum_{j=0}^{\log_b n-1} a^j f(n/b^j)}_{\text{LEVELS}}, \tag{3}$$

where $k = n^{\log_b a}$.

$\boxed{\textbf{Case 1: "Heavy leaves".}}$

- $f(n) = O(n^{k-\varepsilon})$

We will show that the running time is dominated by the LEAVES term $\Theta(n^k)$. Thus we need to show that LEVELS is the same order or lower-order term: LEVELS $\in O(\text{LEAVES})$.

Want to prove LEVELS $\in O(n^k)$**:** From the condition of the case we have

$$\text{LEVELS} = O\left(\sum_{j=0}^{\log_b n-1} a^j (n/b^j)^{k-\varepsilon}\right).$$

Now we have:

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j (n/b^j)^{k-\varepsilon} &= n^{k-\varepsilon} \sum_{j=0}^{\log_b n-1} \frac{a^j}{b^{kj}} \cdot b^{\varepsilon j} && \text{because } 1/b^{k-\varepsilon} = b^{\varepsilon}/b^k \\
&= n^{k-\varepsilon} \underbrace{\sum_{j=0}^{\log_b n-1} b^{\varepsilon j}}_{\text{geometric series}} && \text{because } b^k = b^{\log_b a} = a \\
&= n^{k-\varepsilon} \left(\frac{b^{\varepsilon \log_b n} - 1}{b^{\varepsilon} - 1} \right) \\
&= n^{k-\varepsilon} O(n^{\varepsilon}) && \text{because } b^{\log_b n} = n \\
&= O(n^k)
\end{aligned}$$

$\boxed{\textbf{Case 2: “Heavy levels”.}}$

- $f(n) = \Theta(n^k)$

This is “balanced case” where the work is equally distributed between all levels of the tree. We will show that each level contributes to the total time by the term $\Theta(n^k)$. Since there are $\log_b n = \Theta(\log n)$ levels, the overall time is LEVELS + LEAVES $= \Theta(n^k) + \Theta(n^k \log n) = \Theta(n^k \log n)$.

Want to prove that the time contributed by jth level is $\Theta(f(n))$, i.e. $a^j f(n/b^j) = \Theta(f(n))$:

Note, that

$$a^j f(n/b^j) = \Theta(a^j (n/b^j)^k) = \Theta\left(n^k \left(\frac{a}{b^k} \right)^j \right) = \Theta(n^k) = \Theta(f(n)),$$

because $b^k = b^{\log_b a} = a$.

$\boxed{\textbf{Case 3: “Heavy top”.}}$

- $f(n) = \Omega(n^{k+\varepsilon})$
- Regularity condition: There exists a constant $c < 1$ such that $af(n/b) \le cf(n)$ for all sufficiently large n.

We will show that in this case the running time is dominated by the “combine” step on the top-most level.

Want to prove $T(n) \in \Omega(f(n))$: Note, that $f(n)$ is part of (3) and all terms in (3) are non-negative. Therefore $T(n) \ge f(n)$.

Want to prove $T(n) \in O(f(n))$: First of all, the sum is dominated by LEVELS part (because LEAVES $\in o(f(n))$). From the regularity condition, we have:

$$\text{LEVELS} \le \underbrace{f(n) + cf(n) + c^2 f(n) + \dots}_{\text{geometric series, } c<1} = f(n) \left(\frac{1}{1-c} \right)$$

Therefore $T(n) \in \Theta(f(n))$ under the conditions of case 3.

5.5 Closest-pair problem

[CLRS2 chapter 33.4]

$\boxed{\textbf{Problem:}}$ Given are n points $p_1, p_2, \ldots, p_n$; point p_i has coordinates (x_i, y_i). Find the closest pair of points (smallest Euclidian distance).

Notation: Let $d(p_i, p_j)$ be the distance of points (p_i, p_j):

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Trivial solution: Compare $d(p_i, p_j)$ for all pairs of points and choose the smallest. This solution takes $\Theta(n^2)$.

Better idea: Divide and conquer
(For simplicity, we will assume that all vertices have different x-coordinates.)

- **Divide.** Find a vertical line that separates the set of points into two halves – P_L (left half) and P_R (right half).
- **Conquer.** Find the closest pair both in the left and the right half recursively. Denote δ_L the distance of the closest pair in P_L, and δ_R the distance of the closest pair in P_R.
- **Combine.** We still did not consider pairs of points where one point is on the left side of the line and one point is on the right side. Let δ_M is the distance of the closest pair of such points. Then the minimum distance is $\min\{\delta_L, \delta_R, \delta_M\}$.

How to implement the combine phase?

Option 1: Try to pair all points in P_L with all points in P_R. To do that, we need $(n/2)(n/2) = n^2/4 = \Theta(n^2)$ distance computations. Thus the total time $T(n)$ is:

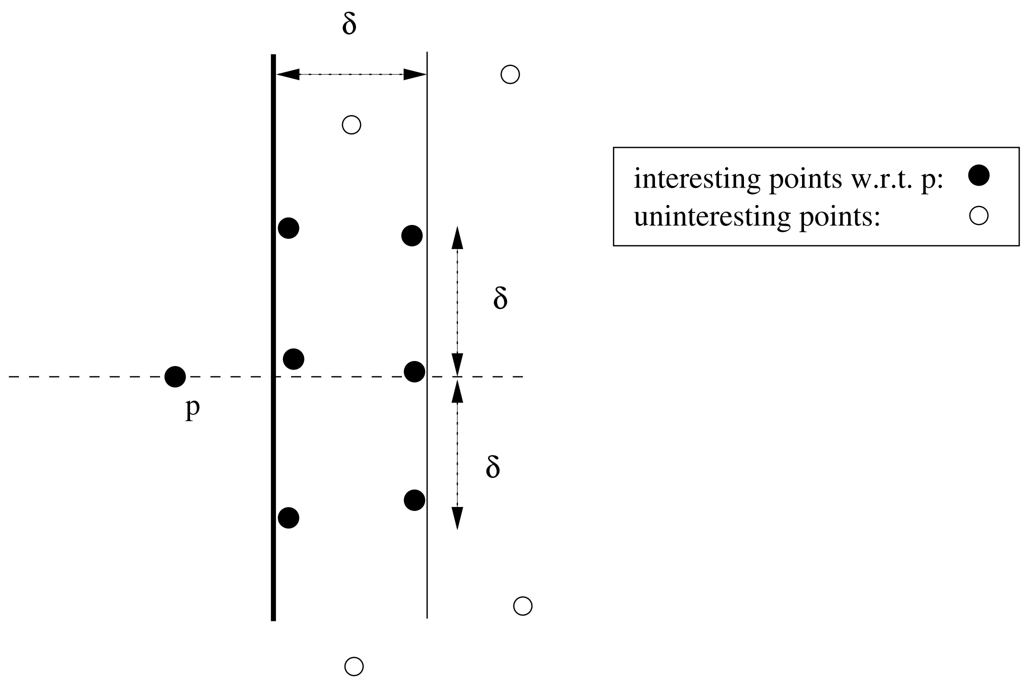
$$T(n) = 2T(n/2) + \Theta(n^2) = \Theta(n^2)$$

(by master theorem) – no good :-(

Option 2: Denote $\delta = \min\{\delta_L, \delta_R\}$. Our objective is to determine, whether there exists a pair of points $p \in P_L$ and $q \in P_R$ such that $d(p, q) < \delta$.

Consider any point $p \in P_L$. Which vertices from P_R may be a "good match" for such vertex and which vertices are "hopeless" (i.e., it is immediately clear that the distance of the vertex from p is longer than δ).

- If p is farther then δ to the left of the splitting line, then there is no $q \in P_R$ such that $d(p, q) \leq \delta$. (It takes more than δ to get only to the splitting line and we did not reach any points yet).
- Similarly, no $q \in P_R$ which is farther thant δ to the right of the splitting line can be succesfully paired with p.
- Finally, if $p = (x, y)$, then only points in P_R with y coordinates in interval $[y - \delta, y + \delta]$ can be succesfully paired with p (others are too far away in the y coordinate).



Natural question: Note that all pairs of points in P_R have mutual distance at least δ. How many of such points you can fit into rectangle $2\delta \times \delta$?

Answer: 6
This means, for every point $p \in P_L$ there are at most 6 points $q \in P_R$ such that $d(p, q)$ can be potentially smaller than δ.

Notation: Let Q_L be the subset of points from P_L which are in the distance at most δ from the splitting line. Similarly, let Q_R be the subset of points from P_R which are in the distance at most δ from the splitting line. Let P be a global array storing all the points.

```
function delta_m(QL,QR,delta)
  // Are there two points p in QL, q in QR such that
  // d(p,q)<=delta? Return closest such pair.

  // Assume QL and QR are sorted by y coordinate
  j:=1; dm:=delta;
  for i:=1 to size(QL) do
    p:=QL[i];
    // find the bottom-most candidate from QR
    while (j<=n and QR[j].y<p.y-delta) do
      j:=j+1;
    // check all candidates from QR starting with j
    k:=j;
    while (k<=n and QR[k].y<=p.y+delta) do
      dm:=min(dm,d(p,QR[k]));
      k:=k+1;

  return dm;
```

```
function select_candidates(l,r,delta,midx)
  // From P[l..r] select all points which are
  // in the distance at most delta from midx line
  create empty array Q;
  for i:=l to r do
    if (abs(P[i].x-midx)<=delta)
      add P[i] to Q;
  return Q;

function closest_pair(l,r)
  // Find the closest pair in P[l..r]
  // assume P[l..r] is sorted by x-coordinate
  if size(P)<2 then return infinity;

  mid:=(l+r)/2; midx:=P[mid].x;

  dl:=closest_pair(l,mid);
  dr:=closest_pair(mid+1,r);
  // as a side effect, P[l..mid] and P[mid+1..r]
  // are now sorted by y-coordinate
  delta:=min(dl,dr);

  QL:=select_candidates(l,mid,delta,midx);
  QR:=select_candidates(mid+1,r,delta,midx);
  dm:=delta_m(QL,QR,delta);

  // use merge as in merge sort to make
  // P[l..r] sorted by y-coordinate
  merge(l,mid,r);

  return min(dm,dl,dr);

//-------- main ---------
// P contains all the points
sort P by x-coordinate;
return closest_pair(1,n);
```

Running time: Let $T(n)$ be the time required to solve the problem for n points.

- Divide: $\Theta(1)$
- Conquer: $2T(n/2)$
- Combine: $\Theta(n)$

So $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$ (by master theorem).

5.6 Linear-time selection

[BB chapter 7.5]

$\boxed{\textbf{Problem:}}$ Given an array of n numbers $A[1..n]$ and a number i ($1 \leq i \leq n$), find the i-th smallest number in A.

Definition: *Median* is the $\lceil n/2 \rceil$-th element in A.

Example: A=(7,4,8,2,4); the 3rd smallest element (and median) is 4.

Trivial solutions:

- In general: Sort the array and take i-th element. $\Theta(n \log n)$
- For i being a small constant (or n minus a small constant): easy linear-time solution. (How would you find a minimum/maximum in the array?)

Idea: Partition-based (divide and conquer) selection

- Choose one element p from array A (*pivot* element)
- Split input into three sets:
  - *LESS* – elements from A that are smaller than p
  - *EQUAL* – elements from A that are equal to p
  - *MORE* – elements from A that are greater than p
- We have three cases:
  - $i \leq |LESS|$: then element we are looking for is also the i-th smallest number in *LESS*,
  - $|LESS| < i \leq |LESS| + |EQUAL|$: then element we are looking for is p,
  - $|LESS| + |EQUAL| < i$: then element we are looking for is also the $(i - |LESS| - |EQUAL|)$-th smallest element in *MORE*.

```
function SELECT(A,i)
  // find i-th element in array A
  p:=choose_pivot(A);

  // partition A into LESS, EQUAL, MORE
  create new arrays LESS, EQUAL, MORE;
  for i:=1 to size(A) do
    if A[i]<p then add A[i] to LESS;
    if A[i]=p then add A[i] to EQUAL;
    if A[i]>p then add A[i] to MORE;

  // decide, what case to pursue
  if size(LESS)>=i then
    return SELECT(LESS,i);
  else if size(LESS)+size(EQUAL)>=i then
    return p;
  else
    return SELECT(MORE,i-size(LESS)-size(EQUAL));
```

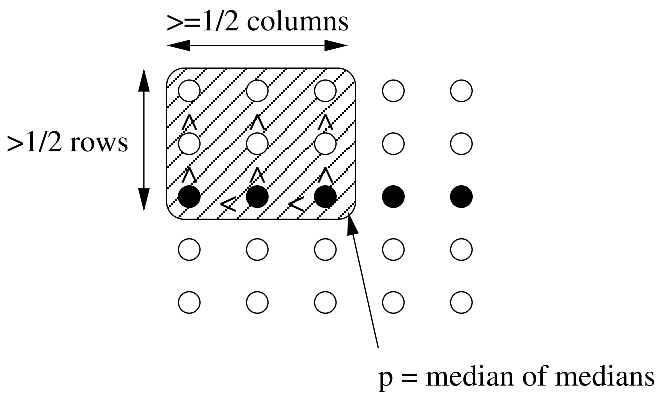
How to choose the pivot?

- **Option 1: arbitrary element (let's say the first).** Assume that we have a sorted array and that we are looking for the n-th smallest element. This will take $\Omega(n^2)$ time.
- **Option 2: random element.** This is better and leads to so called *randomized algorithm.* It can be shown that in such case *expected running time* is $\Theta(n)$, while *worst-case running time* is still $\Theta(n^2)$. See [CLRS, 9.2], if interested.
- **Option 3: "magic".** Use the following "magic" to select the pivot:

  1. Split array $A[1..n]$ into $n/5$ groups with 5 elements each.
  2. From each group select the third smallest element (i.e., take median of each of the groups). Denote set of these elements *MEDIANS*.
  3. Recursively call `SELECT` to obtain median of *MEDIANS* (i.e., $\lceil n/2 \rceil$-th smallest element of *MEDIANS*).
  4. Take the resulting element as pivot p.

  Lemma 5. *At least* $1/4$ *of elements in* A *are smaller than or equal to* p*.*

  Proof. Imagine sorting elements in each of the groups from smallest to largest and ordering groups by their median (this is not done by the algorithm). Let us represent the whole set A by a table where each group is depicted as a single column and the columns are ordered by their medians. Then the following figure demonstrates the claim (for the case where n is a multiple of 5):

  □

  Similarly:

  Lemma 6. *At least* $1/4$ *of elements in* A *is greater than or equal to* p*.*

  The corollary below follows immediately from the two lemmas above.

  Corollary 1. *If we use* p *as pivot in* `SELECT`*, than arrays LESS and MORE have at most* $3/4 \cdot n$ *elements.*

Running time: The running time $T(n)$ of the `SELECT` algorithm with such selection of pivot can be derived as follows:

- Divide phase: $\Theta(n)$
- Conquer phase:
  - to select "median of medians": $T(\lceil n/5 \rceil)$;
  - to run selection on one of the *LESS* or *MORE*: $\leq T(\lfloor 3/4 \cdot n \rfloor)$
- Combine phase: no work

Thus we have: $T(n) = T(\lceil n/5 \rceil) + T(\lfloor 3/4 \cdot n \rfloor) + \Theta(n)$, $T(1) = \Theta(1)$.

Lemma 7. *$T(n) \leq c'n$ (constant c' to be determined later)*

Proof. By induction on n (substitution method).

- **Base case.** For $n < 40$ the claim clearly holds as far as c' is large enough.
- **Induction step.** Assume that $n \geq 40$ and for all $n' < n$, $T(n') \leq c'n'$. Then:

$$\begin{aligned}
T(n) &\leq T(\lceil n/5 \rceil) + T(\lfloor 3/4 \cdot n \rfloor) + cn \\
&\leq c'(n/5 + 1) + 3/4 \cdot c'n + cn \\
&\leq 1/5 \cdot c'n + 1/40c'n + 3/4 \cdot c'n + cn \\
&\leq 39/40 \cdot c'n + cn \\
&\ \text{as far as } c < 1/40 \cdot c' \text{:} \\
&\leq c'n
\end{aligned}$$

□

Thus the running time of the `SELECT` algorithm is $\boxed{O(n).}$

What is this good for?

Recall QuickSort. The QuickSort works as follows:

- Select pivot element p.
- Split array into two parts: elements smaller than p and elements larger than p.
- These can be sorted separately.

Hopefully, whenever we split the array, we get subarrays of approximately same size and thus achieve $\Theta(n \log n)$ running time. However, if we are unlucky, we get $\Omega(n^2)$ running time.

Idea: What if we use our `SELECT` algorithm to select pivot p to be median? Then every time we split, we guarantee "almost equal" split thus having $\Theta(n \log n)$ worst-case running time.

This is quite slow in practice – the constant associated with running `SELECT` is too large. It is better just select a random element (it can be proved that then we get $\Theta(n \log n)$ expected running time).

5.7 Making divide and conquer algorithms faster in practice

Divide and conquer algorithms have often large multiplicative and additive constant overhead (for recursion, etc.) which makes them slower for small size data sets than the trivial algorithm.

Running time of a divide and conquer algorithm can be reduced if we solve small subproblems by the trivial algorithm (instead of dividing them further). When to "divide" and when to use trivial algorithm needs to be determined empirically.

Example:

```
function SELECT(A,i)
* if size(A)<100 then
*   sort elements of A;
*   return A[i];
  else
    // find i-th element in array A
    p:=choose_pivot(A);

    // partition A into LESS, EQUAL, MORE
    create new arrays LESS, EQUAL, MORE;
    for i:=1 to size(A) do
      if A[i]<p then add A[i] to LESS;
      if A[i]=p then add A[i] to EQUAL;
      if A[i]>p then add A[i] to MORE;

    // decide, what case to pursue
    if size(LESS)>=i then
      return SELECT(LESS,i);
    else if size(LESS)+size(EQUAL)>=i then
      return p;
    else
      return SELECT(MORE,i-size(LESS)-size(EQUAL));
```

Designing efficient algorithms – summary

$\boxed{\textbf{Greedy algorithms}}$

- take locally optimal choice in each step
- easy to design and implement,
- usually efficient,
- often greedy approach cannot be used,
- main challenge: prove correctness.

Solved problems:

- Activity selection $\Theta(n \log n)$
- Construction of Huffman trees $\Theta(n \log n)$
- Coin changing (some money systems) $\Theta(m)$

$\boxed{\textbf{Dynamic programming}}$

- compute values for subproblems organized in a big matrix
- easy to implement
- main challenge: come up with the right subproblem.

Solved problems:

- General coin changing $\Theta(mS)$
- Longest common subsequence $\Theta(mn)$
- Knapsack problem $\Theta(nW)$
- Shortest triangulation $\Theta(n^3)$

$\boxed{\textbf{Divide and conquer}}$

- divide problem into subproblems, solve them recursively and combine partial solutions
- efficient
- sometimes difficult to implement and large overhead
- main challenge: running time analysis

Solved problems:

- Sorting (merge sort, quick sort) $\Theta(n \log n)$
- Multiplying large number $\Theta(n^{1.58\cdots})$
- Matrix multiplication (Strassen's algorithm) $\Theta(n^{2.81\cdots})$
- Closest pair problem $\Theta(n \log n)$
- Linear-time selection $\Theta(n)$