Introduction to Algorithms

7. More advanced topics

Manuel – Fall 2020

Chapter organisation

More advanced topics

Amortized analysis

Approximation algorithms

Multi-threaded algorithms

Solving hard problems

Given an $\mathcal{NP}$-complete problem:

- It is impossible to efficiently find a solution
- In practice it might have many applications

Hope for such problems:

- Inputs are always small
- Specific sub-cases arising in practice can be solved efficiently
- An almost optimal solution is sufficient and can be computed efficiently

Approximation algorithms

Definition

1. For a given problem P, an algorithm returning a near-optimal solution to P is called an *approximation algorithm*.
2. Let the cost of an optimal solution be C^* and the one of an approximation be C. If for any input of size n there exists an *approximation ratio* $\rho(n)$, such that

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n),$$

then the algorithm is said to be a $\rho(n)$-*approximation algorithm*.

Approximation algorithms

Remark.

- The approximation ratio can never be less than 1
- Approximation algorithms are expected to
  - Be polynomial time
  - Feature slowly growing approximation ratio as n increases
- Some approximation algorithms take an input parameter defining the precision of the approximation. The more precise the approximation, the longer the running time.

The vertex-cover problem

Problem (Optimal Vertex Cover)

Let $G = \langle V, E \rangle$ be an undirected graph. A *vertex cover* is a subset $V' \subseteq V$ such that if $(u, v) \in E$ then at least u or v is in V'. Find a vertex cover of minimum size.

The vertex-cover problem

Problem (Optimal Vertex Cover)

Let $G = \langle V, E \rangle$ be an undirected graph. A *vertex cover* is a subset $V' \subseteq V$ such that if $(u, v) \in E$ then at least u or v is in V'. Find a vertex cover of minimum size.

Algorithm. (*Approx Vertex Cover*)

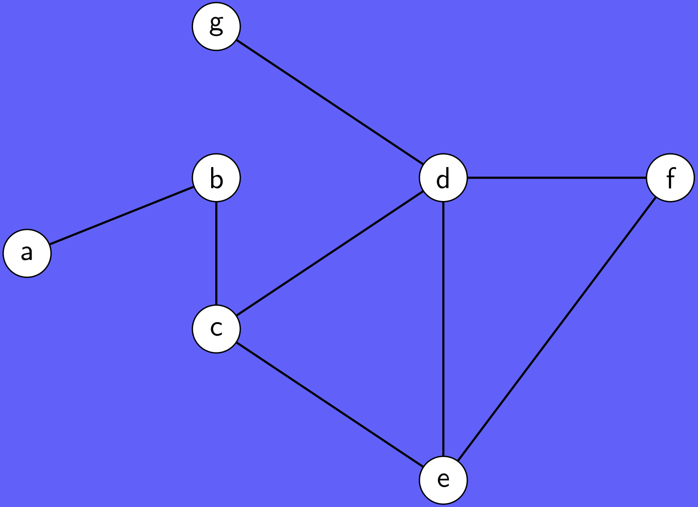
Input : $G = \langle V, E \rangle$ an undirected graph

Output : C a nearly optimal vertex cover

```
$C \leftarrow \emptyset$; $E' \leftarrow G.E$;
while $E' \neq \emptyset$ do
    $e \leftarrow$ an arbitrary edge of $E'$; $C \leftarrow C \cup \{e.u, e.v\}$;
    remove from $E'$ any edge incident to $u$ or $v$;
end while
return $C$;
```

Approximating vertex-cover

Exercise.



Approximating vertex-cover

Theorem

Approx Vertex Cover runs in time $\mathcal{O}(|V| + |E|)$ and is a 2-approximation algorithm.

Proof. Representing E' using an adjacency list leads to $\mathcal{O}(|V| + |E|)$.

First note that C is a vertex cover since the algorithm loops until all the edges from E have been covered by some vertex in C.

Let A denote the set of all the edges selected by the algorithm. Then any cover includes at least one endpoint, and by construction no two edges in A share a common endpoint. Thus there is no two edges in A that are covered by the same vertex in C^*, and we have the lower bound $|C^*| \geq |A|$.

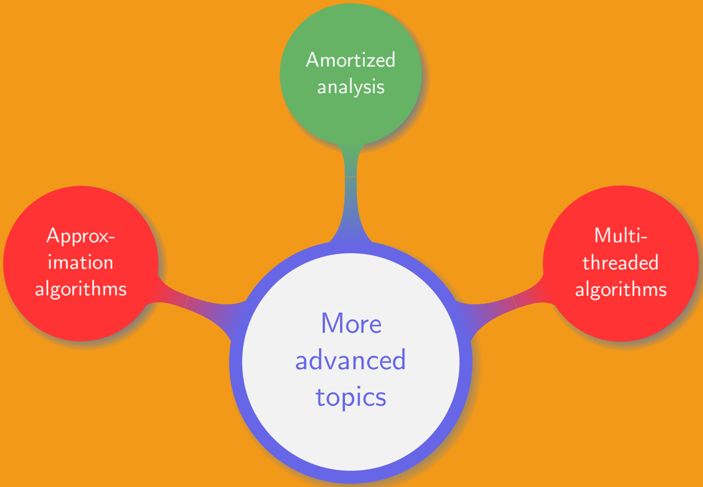
Finally noting that $|C| = 2|A|$ yields $|C| \leq 2|C^*|$. □

Quality of the approximation

Common methodology for approximation algorithms:

- An optimal solution is unknown
  e.g. the size of the vertex cover is unknown
- Determine a lower bound on an optimal solution
  e.g. in approx vertex cover (7.6) the set of the selected edges forms a maximum matching (4.42), which provides a lower bound on the size of an optimal vertex cover
- Relate the size of the approximation to the lower bound on the optimal solution
  e.g. the approximation ratio is obtained by relating $|C|$ to $|A|$

Chapter organisation



More advanced topics

Amortized analysis

Approx-imation algorithms

Multi-threaded algorithms

Motivating problem

A *dynamic table* is an array which automatically resizes as elements are added or removed. A common strategy consists in doubling the size of the table as soon as an overflow occurs. In such a context we want to determine the cost of n insertions.

We define the cost c_i of the ith operation to be i, if $i-1$ is a power of 2, and 1 otherwise. The cost C_n of n insertions is given by

$$\begin{aligned} C_n &= \sum_{i=1}^{n} c_i \\ &\leq n + \sum_{j=0}^{\lfloor \log(n-1) \rfloor} 2^j \\ &\leq 3n = \Theta(n) \end{aligned}$$

Hence the average cost for each operation is $\Theta(n)/n = \Theta(1)$.

Amortized analysis

Definition (Amortized analysis)

Given a sequence of operations, an *amortized analysis* is a strategy allowing to show that the average cost per operation is small although single operations in the sequence might be expensive.

Remark. Amortize analysis:

- Does not uses probabilities
- Evaluates the average performance of each operation in the worst case
- Provides a more precise and useful evaluation of the difficulty of a problem than average case complexity

Common strategies

Three main approaches to amortized analysis:

- *Aggregate method:* most simple approach where the total running time for the sequence is analysed and divided by the number of operations
- *Accounting method:* charge each type of operation a constant cost such that the extra charge on inexpensive ones can be stored in a "bank" and used to pay expensive subsequent operations
- *Potential method:* evaluate and store the total amount of extra work done over the whole data structure and release it to cover the cost of subsequent operations

Remark. The average cost for each operation in the case of dynamic tables (7.11) was determined using the aggregate method.

Accounting method

Let c_i denote the actual real cost of the ith operation, while $\hat{c}_i$ represents its amortized cost, i being larger than 1. The goal is to always ensure that at any stage the sum of the $\hat{c}_i$ is larger than the sum of the c_i, meaning that some extra credit is available for future operations.

For a sequence of n operations the bank balance never becoming negative can be expressed as

$$\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0.$$

Example. In the case of dynamic tables (7.11) charge $\hat{c}_i = 3$ units for each insertion: one is used on insertion while the 2 remaining are saved for a future use. In particular when memory is reallocated a unit is use for reassigning the current element while the last one is spent to move an older value.

Potential method

Let D_0 be an initial data structure. At step i, the ith operation, costing c_i, is applied to D_{i-1}. The *potential* associated with the data structure D_i, denoted $\Phi(D_i)$, is a real number and Φ is called the *potential function*. The amortized cost corresponds to the actual cost plus the change in potential induced by the operation. This formalizes as $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$, and we get

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \geq \sum_{i=1}^{n} c_i\end{aligned}$$

Example. For the dynamic tables (7.11) set the potential after the ith insertion to $\Phi(D_i) = 2i - 2^{\lceil \log_2 i \rceil}$. Then whether or not $i-1$ is a power of 2, the amortized cost is 3.

Notes

Remarks on the three methods:

- The accounting and potential methods are more powerful and refined than the aggregate one; they are equivalent in terms of applicability and precision of the bound provided
- The accounting method charges a different cost for each type of operation
- The potential method focuses on the effect of a particular operation at a specific time, and in particular on the cost of future operations
- Different methods might lead to different bounds, but always upper bound the actual cost

Union-Find data structure

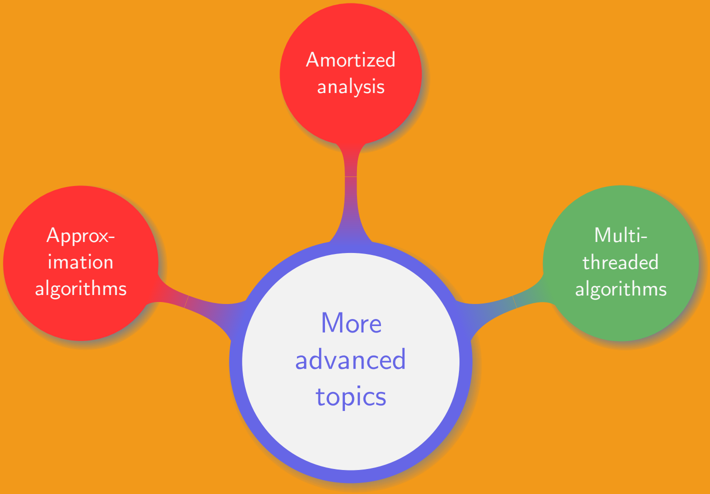
For the Union-Find data structure (1.37), theorem 2.27 can be proven using either the accounting or the potential method.

The potential method suits this example since the goal is to determine how fast the tree is flattening, or in other words how each operation affects the whole data structure. By observing the change in potential after each type of operation it is then possible to determine the amortized cost associated with each of them.

The first and less straight forward step consists in properly setting the potential function. Two cases have to be considered (i) x is a root or has rank 0, and (ii) x is not a root and has rank larger or equal to 1. Defining the potential of x after i operations is most complicated in the latter case.

Once the potential function has been properly defined it can be bounded by $0 \leq \phi_i(x) \leq \alpha(n) \cdot x.rank$, and the amortized cost of the various operations can be determined.

Chapter organisation



More advanced topics

Amortized analysis

Approx-imation algorithms

Multi-threaded algorithms

A new model

Algorithm discussed so far targeted *uniprocessor* computers, while most modern systems feature *multiprocessors* sharing a common memory. The question is then to know how to adapt those algorithms to this new context, and in particular how to efficiently partition the work among several *threads*, each having roughly the same load.

The most simple strategy consists in employing a software layer, called *concurrency platform*, which coordinates, schedules, and manages the resources.

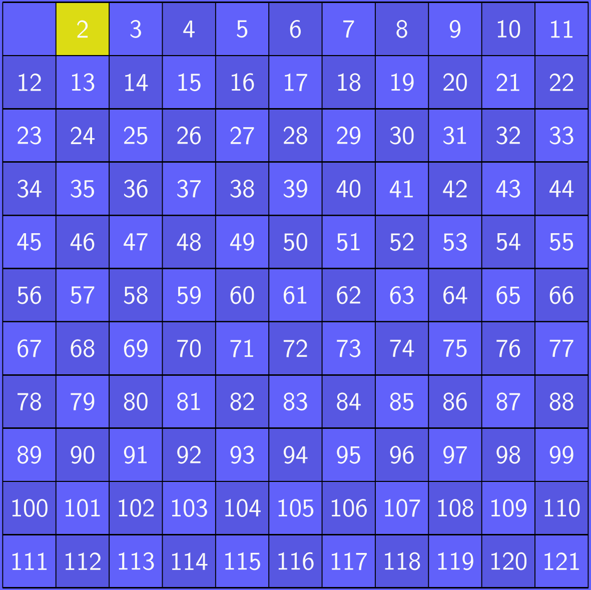
A simple extension to *serial programming* is the addition of the following instructions: `parallel`, `spawn`, and `sync`.

Concurrency instructions

The three new instructions:

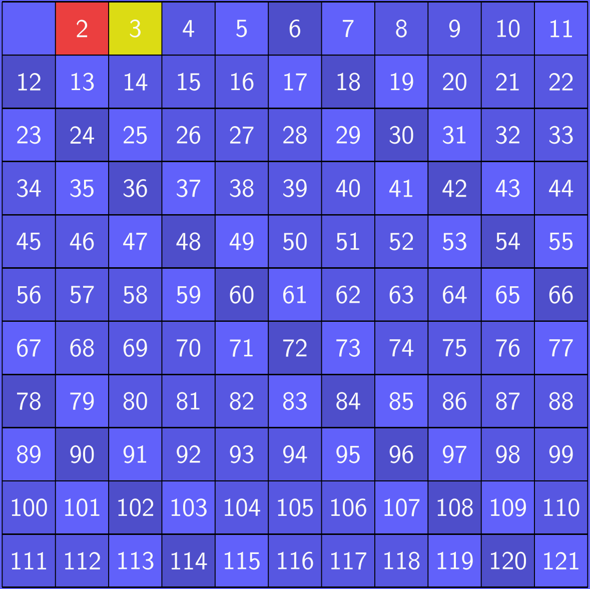
- `parallel:` added to loops to indicate that iterations can be computed in parallel
- `spawn:` executes a new parallel process, the current one may then choose to run concurrently or wait for its child
- `sync:` requests the process to wait for all spawned processes to complete

Sieve of Eratosthenes



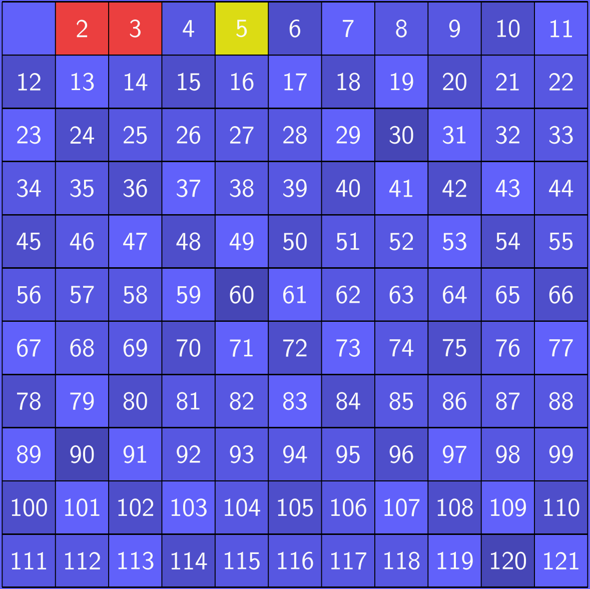
	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22
23	24	25	26	27	28	29	30	31	32	33
34	35	36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63	64	65	66
67	68	69	70	71	72	73	74	75	76	77
78	79	80	81	82	83	84	85	86	87	88
89	90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120	121

Sieve of Eratosthenes



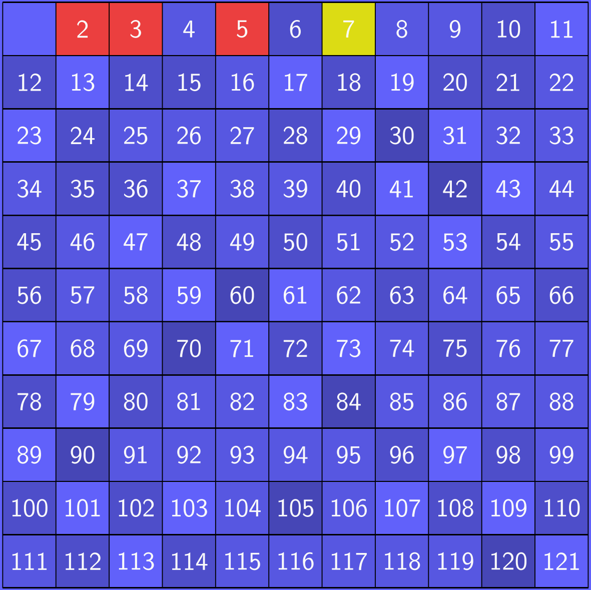
	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22
23	24	25	26	27	28	29	30	31	32	33
34	35	36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63	64	65	66
67	68	69	70	71	72	73	74	75	76	77
78	79	80	81	82	83	84	85	86	87	88
89	90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120	121

Sieve of Eratosthenes



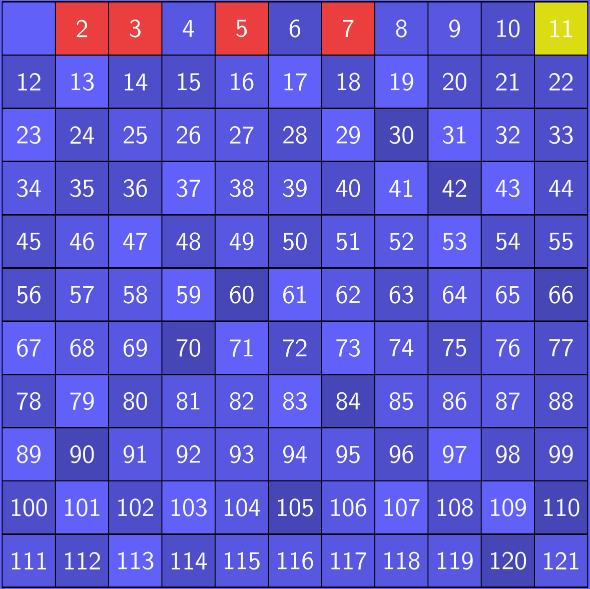
	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22
23	24	25	26	27	28	29	30	31	32	33
34	35	36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63	64	65	66
67	68	69	70	71	72	73	74	75	76	77
78	79	80	81	82	83	84	85	86	87	88
89	90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120	121

Sieve of Eratosthenes



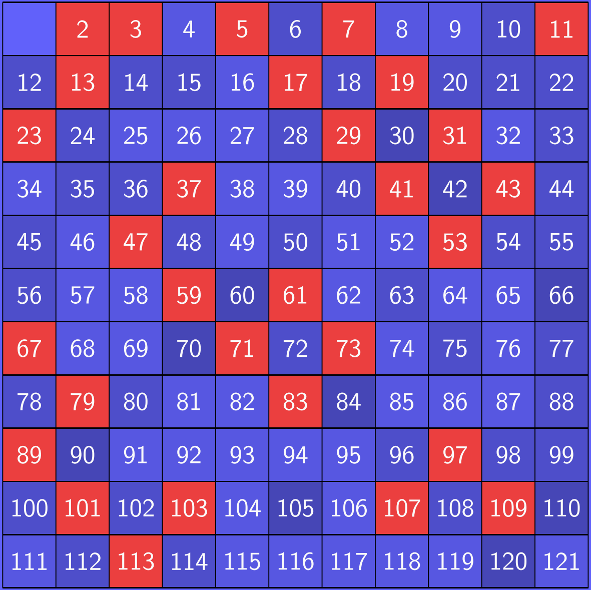
	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22
23	24	25	26	27	28	29	30	31	32	33
34	35	36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63	64	65	66
67	68	69	70	71	72	73	74	75	76	77
78	79	80	81	82	83	84	85	86	87	88
89	90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120	121

Sieve of Eratosthenes



	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22
23	24	25	26	27	28	29	30	31	32	33
34	35	36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63	64	65	66
67	68	69	70	71	72	73	74	75	76	77
78	79	80	81	82	83	84	85	86	87	88
89	90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120	121

Sieve of Eratosthenes



	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22
23	24	25	26	27	28	29	30	31	32	33
34	35	36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63	64	65	66
67	68	69	70	71	72	73	74	75	76	77
78	79	80	81	82	83	84	85	86	87	88
89	90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120	121

Parallel threads

In Eratosthenes sieve all the loop iterations are independent from each others. It is therefore extremely simple to parallelize the algorithm only using the `parallel` keyword in conjunction with the `for` loop.

In other cases a *race condition* might occurs, leading to a result that is non-deterministic. A simple example is as follows.

Algorithm.

```
Input  : $x$
Output: $x + 2$ is expected
parallel for $i \leftarrow 1$ to $2$ do
    $x \leftarrow x + 1$;
end for
return $x$;
```

Modeling multi-threaded algorithms

Two main strategies can be employed to avoid race conditions: (i) a thread sets a *lock* when reaching a critical part of the code to prevent any other thread to run it at the same time; (ii) use special hardware instructions called *atomic operations* which can run several operations at once.

Definition (Computation DAG)

Multi-threaded algorithms can be represented as Directed Acyclic Graph (DAG), often referred to as *computation dag*.

More specifically each vertex stands for an instruction while the edges organize the dependencies between the various instructions. An edge (u, v) implies that instruction u must be run before v.

A chain composed of one or more instructions and not containing any concurrency instruction is called a *strand*. Two strands connected by a directed path are in *series*, or otherwise in *parallel*.

Generalities

A computer composed of n processors is expected to run n concurrent threads. The *scheduler* is the part of the Operating System which decides which thread to run and on which CPU.

Several approaches are to be used depending on the hardware. Modern desktop and laptop computers feature a memory shared among several CPU, which is much easier to handle than having an independent memory for each CPU.

Example. OpenMP allows to easily add parallelism to existing source code without requiring any significant rewrite. It perfectly suites systems where the memory is shared among multiple CPUs.

MPI offers advanced possibilities more specifically targeting systems with a distributed memory. It is very common in the realm of high end computing, especially on clusters. MPI often requires a complete redesign, or even change of algorithm to implement.

Merge Sort

Assuming the existence of a `Merge` function, the following algorithm performs serial Merge Sort on a given list L.

Algorithm. (*Merge Sort*)

Input : $L = a_1, \dots, a_n$

Output : L, sorted into elements in non-decreasing order

```
Function MergeSort($L$):
    if $n > 1$ then
        $m \leftarrow \lfloor n/2 \rfloor$;
        $L_1 \leftarrow a_1, \dots, a_m$; $L_2 \leftarrow a_{m+1}, \dots, a_n$;
        $L \leftarrow$ Merge(MergeSort($L_1$), MergeSort($L_2$))
    end if
    return $L$
end
```

The recursive calls `MergeSort` seem to make this algorithm a good candidate for parallelism.

Merge

Algorithm. (*Merge*)

Input : L_1, L_2, two sorted lists

Output : L a merged list of L_1 and L_2, with elements in increasing order

```
Function Merge($L_1, L_2$):
    $L \leftarrow \emptyset$;
    while $L_1 \neq \emptyset$ and $L_2 \neq \emptyset$ do
        $x \leftarrow \min(L_1, L_2)$;        /* smallest element in $L_1$ and $L_2$ */
        append $x$ to $L$;
        if $L_1 = \emptyset$ or $L_2 = \emptyset$ then
            $L_1 = \emptyset$ ? append $L_2$ to $L$ : append $L_1$ to $L$;
        end if
    end while
    return $L$;
end
```

Parallel Merge Sort

Algorithm. (*Parallel Merge Sort*)

Input : $L = a_1, \ldots, a_n$
Output : L, sorted into elements in non-decreasing order

```
Function P-MergeSort($L$):
    if $n > 1$ then
        $m \leftarrow \lfloor n/2 \rfloor$;
        $L_1 \leftarrow a_1, \ldots, a_m$; $L_2 \leftarrow a_{m+1}, \ldots, a_n$;
        $L_1 \leftarrow$spawn P-MergeSort $(L_1)$;
        $L_2 \leftarrow$spawn P-MergeSort $(L_2)$;
        sync $L \leftarrow$Merge $(L_1, L_2)$;
        return $L$;
    end if
end
```

In fact this parallel version does not improve much on the serial version: although Merge Sort is parallel, Merge remains serial and as such the recurrence relation still features the same $\mathcal{O}(n)$ term. Moreover the parallelism is only $\Theta(\log n)$.

Parallel Merge

Although at times serial algorithms can easily be made parallel, it is often impossible, and they require to be completely redesigned using a totally different approach. In some other cases algorithms might seem to be intrinsically serial, but can still be adjusted at the cost of some extra work. This is for instance the case of the Merge algorithm.

The idea is to carefully generate four lists that can be merged two by two without altering the ordering. The process is as follows.

1. Find the median element in the longest list: define two sublists $L_{1,L}$ and $L_{1,R}$
2. Determine its corresponding potential location in the second list: define two sublists $L_{2,L}$ and $L_{2,R}$
3. Recursively merge the lists $L_{1,L}$ and $L_{2,L}$ and $L_{1,R}$ and $L_{2,R}$

Parallel Merge

Algorithm. (*Parallel Merge*)

Input : L_1, L_2, two sorted lists, with $L_1.length > L_2.length$
Output : L, a merged list of L_1 and L_2, with elements in increasing order

```
Function P-Merge($L_1, L_2$):
    if $L_2 = \emptyset$ then $L \leftarrow L_1$ ;
    else
        $m_1 \leftarrow \lfloor L_1.length/2 \rfloor$;
        $L_{1,L} \leftarrow \left\{L_{1_i}\right\}_{1 \leq i < m_1}$; $L_{1,R} \leftarrow \left\{L_{1_i}\right\}_{m_1 < i \leq L_1.length}$;
        $m_2 \leftarrow$ index where $m_1$ would be in $L_2$;
        $L_{2,L} \leftarrow \left\{L_{2_i}\right\}_{1 \leq i < m_2}$; $L_{2,R} \leftarrow \left\{L_{2_i}\right\}_{m_2 \leq i \leq L_2.length}$;
        $L_L \leftarrow$ spawn P-Merge($L_{1,L}, L_{2,L}$); $L_R \leftarrow$ P-Merge($L_{1,R}, L_{2,R}$);
        sync;
        $L \leftarrow$ concatenate $L_L$, $L_{1_{m_1}}$, $L_R$;
    end if
    return $L$;
end
```

The complexity is $\Theta(n)$ while the parallelism grows to $\Theta(n/\log^2 n)$.

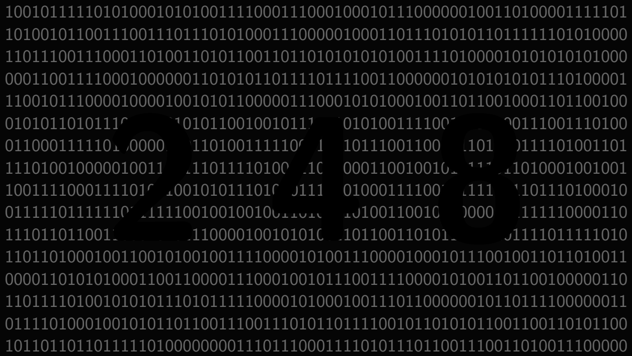
Final notes

Designing efficient algorithms requires:

- A good understanding of the problems
- An advanced knowledge of the underlying mathematics
- Much time spent on a trial and errors approach
- The consideration of all the corner cases

Key points

- When to use approximation algorithms?
- How to evaluate the quality of an approximation algorithm?
- Describe amortized analysis.
- Cite the three main methods to perform amortized analysis.
- What are the instruction `parallel`, `spawn`, and `sync`?



Thank you!