Several improvements about BKZ algorithm

Ziyu Zhao

Tsinghua University

Abstract. todo

1 Introduction

Lattice based cryptography is nowadays a important part of post-quantum cryptography because it's fast and widely used in fully homomorphic encryption. The security of it mainly based on some lattice problems, such as learning with error problem (LWE) and ntru problem. These problems can be reduce to approximate shortest vector problem (ASVP), i.e. to find a relatively short vector given a lattice basis. And currently lattice reduction is the most efficient way to solve such problems, thus it is important to know the concrete hardness of ASVP.

Currently there are two types of algorithm to find short vector in the lattice given a lattice basis. One is SVP algorithms like enumeration and sieving, which can find almost the shortest vector in the lattice but the cost is exponential in the dimension of lattice. These SVP algorithms can only be applied on lattice with a small dimension. Another type of algorithm, for instance LLL algorithm and BKZ algorithm can work on high dimensional lattice in a realistic time. LLL algorithm is extremely fast and often used as preprocessing, BKZ algorithm gives a bridge from shortest vector in small dimension to short vector with the same root hermite factor in high dimension.

BKZ algorithm were first purposed by Schnorr in the 80's. It does enumeration on local blocks to find short vector then insert the new vector in the basis. Larger local blocksize gives shorter vector and cost more time. In 2011, Nguyen used some pruning technique in the enumeration step, it makes BKZ algorithm with a higer local blocksize practicable. In 2016, Yuntao Wang et al. purposed progressive BKZ, it start with a small blocksize, and increase it in a well organized manner, makes the algorithm significantly faster.

In this paper, we will give several further improvements of BKZ algorithm, and applied this techniques in lattice reduction with a SVP subroutine based on enumeration (these techniques can also be applied to sieving based SVP subroutine well). We implemented the new BKZ algorithm and tested it on several Ideal lattice challenges (the ideal lattice structure is never used), the running result shows we get a speed up with a factor $2^{3\sim4}$, which may be further improved since we did not used a well organized progressive BKZ (our blocksize are simply $80, 88, 96, \cdots$). Moreover our new BKZ algorithm is still easy to simulate (as BKZ 2.0) if we know the behavior of the SVP subroutine well.

Road map. In section 2, we present some basic facts about lattice and introduce the notations. Then in section 3, we will recall the developments of BKZ algorithm in the history. Our futher improvements of BKZ will be given in section 4. The infomation about the lattice challenge and the simulation of BKZ algorithm is in section 5.

2 Preliminaries

Lattice is discrete subgroup in $\mathbb{R}^m$. A lattice L admits a integral basis $\mathbf{B} = \{\mathbf{b}_1, \mathbf{b}_2, \cdots, \mathbf{b}_n\}$ such that each vector $\mathbf{v}$ in L can be represented uniquely as linear combination with integral coefficients of $\mathbf{B}$. We say n is the dimension of the lattice. The determinant of the lattice is defined to be $\sqrt{\det(\mathbf{B}\mathbf{B}^T)}$ which is equal to the absolute value of $\det(\mathbf{B})$ if $m = n$.

Gaussian Heuristic Because of the discreteness, the shortest vector in L exists (not unique in general). It is extremely hard to compute the shortest vector (proved to be NP-hard problem), but the length of it is estimated to be

$$\frac{\Gamma(\frac{n}{2}+1)^{\frac{1}{n}}}{\sqrt{\pi}} \cdot \det(L)^{\frac{1}{n}} \approx \sqrt{\frac{n}{2\pi e}} \cdot \det(L)^{\frac{1}{n}}$$

when the lattice is *random* and n is not too small. In practice it works well if $n \geqslant 40$.

Gram-Schmidt orthogonalization The Gram-Schmidt orthogonalization of $\mathbf{B}$ is given by $\mathbf{B}^* = (\mathbf{b}_1^*, \cdots, \mathbf{b}_n^*)$ where $\mathbf{b}_i^*$ is defined by

$$\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{ij} \mathbf{b}_j^*, \ \mu_{ij} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\|\mathbf{b}_j^*\|^2}$$

we further denote by B_i the square of $\|\mathbf{B}_i^*\|$, we will call $[B_1, B_2, \cdots, B_n]$ the distance vector of the basis $\mathbf{B}$. The distance vector of a basis contains lots of infomation about this notation will be heavily used in the analysis of BKZ algorithm. we should also introduce the concept of local projected lattice. Let π_i be the orthogonal projection to $\mathrm{span}(\mathbf{b}_1, \cdots, \mathbf{b}_{i-1})^{\perp}$. Then we define the local projected lattice $L_{[i,j]}$ to be the lattice spanned by $B_{[i,j]} = (\pi_i(\mathbf{b}_i), \cdots, \pi_i(\mathbf{b}_j))$.

Root Hermite Factor For a vector $\mathbf{v}$ in a n dimensional lattice L, we define the root Hermite factor to be

$$\delta = \mathrm{rHF}(\mathbf{v}) = \frac{\|\mathbf{v}\|}{\det(L)}^{\frac{1}{n}}$$

the root Hermite factor measures the quality of the vector. The hardness to get a vector of certain length mainly depends on its root Hermite factor.

3 history of BKZ algorithm

3.1 the original algorithm

The first version of BKZ algorithm were purposed by Schnorr and Euchner as a generalization of the famous LLL algorithm. Briefy, LLL algorithm gives the basis an order, then always reduce the latter vector by the former ones, and after the reduction done, it trys to make the former one shorter (in the local projected lattice) by swaping contiguous vector pairs. The algorithm terminates when no more swap or reduction can be done. The first vector of the output basis is of length about 1.02^n times the Gaussian heuristic in practice, and the running time is polynomial in n, where n is the dimension of the lattice.

BKZ algorithm replace the *swap* in LLL algorithm by a full enumeration in the local projected lattice to get shorter vector. This vector will be inserted in the basis in a preselected place, and we use a LLL algorithm to remove the linear dependency. The size of the local projected lattice is fixed and the place to do enumeration is pre-specified. Same as LLL algorithm, BKZ algorithm terminates when no notrivial insertion can be done. The algorithm works as following:

Algorithm 1: BKZ algorithm

Input: a basis $B = (\mathbf{b}_1, \cdots, \mathbf{b}_n)$, blocksize d
Output: A BKZ-d reduced basis
LLL(B); **while** *last epoch did a nontrivial insertion* **do**
  for $i = 1, 2, \cdots, n-1$ **do**
    $h = \max(i + d - 1,\, n)$;
    $\mathbf{v} = \text{full\_enum}(L_{[i,h]})$; //find shortest vector by enumeration
    if $\|\mathbf{v}\| < \|\mathbf{b}_i\|$ **then**
      LLL$(\mathbf{b}_1, \cdots, \mathbf{b}_{i-1}, \mathbf{v}, \mathbf{b}_i, \cdots, \mathbf{b}_{\max(h+1,n)})$;
    else
      LLL$(\mathbf{b}_1, \cdots, \mathbf{b}_{\max(h+1,n)})$;

The running time of BKZ algorithm increase while the blocksize increase. It is not proved to be polynomial in n (the dimension) for fixed blocksize, even though for small blocksize d (for example $d < 20$), the algorithm always terminates in reasonable time and the output quality is significantly improved. For $d = 20$ and n sufficiently large, the shortest vector it founds has length around 1.0128^n times the Gaussian heuristic.

3.2 BKZ2.0

In 2011, Yuanmi and Nguyen gives several improvements of the original BKZ algorithm, which makes BKZ algorithm with a high blocksize ($d \sim 100$) practicable. The root hermite factor of the output vector is now improved to about 1.0095. They mainly did the following:

The first point is that, for a large blocksize d ($d \geqslant 40$), before no new insertion can be done, there is a long time that the quality of the basis improves poorly. So they used the *early abort* technique to stop the algorithm as soon as the quality of the basis not improve further. This provides an exponential speed up in practice without degenerating the output quality.

They also did some modification on the enumeration step. For a larger blocksize d, experiments shows the running time of BKZ is dominanted by the enumeration subroutine. They used pruned enumeration instead of the full enumeration, a proper pruning can gives an exponential speed up ($2^{d/4}$) while still output the shortest vector with a high probablity. They further gives a *extreme pruning* technique, which repeat a further pruned enumeration which output the shortest vector with a low probablity for several times. This leads to a speed up of $2^{d/2}$.

Another thing they did is to preprocessing the basis before the enumeration, since the nodes to enumerate will be fewer if the basis have a better quality. Taking some time to do a light reduction will largely reduce the enumeration time, in BKZ 2.0 they choose a BKZ algorithm with a small blocksize as preprocessing.

3.3 progressive BKZ

Progressive BKZ mainly means to progressively enlarge the blocksize while doing reduction. The key idea is if a enumeration with low dimension can further reduce the lattice, there is no need to use a much larger dimension since the cost for SVP is at least exponential in the dimension. This technique were mentioned in several studies including. These works mainly different in the way they increase the blocksize. In 2016, did a precise Cost Estimation of the progressive BKZ, and gives a optimized blocksize strategy. In their extimation, to do a BKZ-100 in a 800 dimensional lattice, their progressive BKZ is $2^2.7$ times faster than BKZ 2.0. And it's extimated 50 times faster than BKZ 2.0 to solve SVP challenge up do dimension 160.

4 Several improvements about BKZ algorithm

In these section we will give several techniques to further accelerate the BKZ algorithm. These techniques can be used to BKZ based on both sieving and enumeration type SVP subroutine. And one can use it almost for free (except for large final run for sieving, which requires more memory) to get a considerable acceleration in practice.

4.1 local basis processing instead of insertion

Currently We have two types of SVP algorithm, enumeration and sieving. Sieving is faster but requires large space which grows exponential in the sieving dimension. To do a large sieve or enumeration with the hope to find the shortest vector is generally not the best choice. The nodes to enumerate grows as the basis get worse, so we always do some preprocessing as in BKZ2.0. So the whole enumeration process not only gives a short vector, but also a rather good basis. For sieving, we often use the left progressive sieve to accelerate, whose speed also relies on the quality of the basis. And we need the first several entries in the distance vector to be short to get a large dimension for free, which saves both time and memory. Thus it will not lead to much further cost to get a good basis.

Only insert one short vector like the original BKZ algorithm or BKZ2.0 will waste the almost *free* basis. It's generally better to compute the transform matrix of local processing (on the local projected lattice) then apply it on the vectors of the original basis (succeed by a size reduction). Then the next local basis to apply the SVP algorithms is already only little bit worse than an HKZ reduced basis. A obvious gain is we need no more preprocessing for it. And this is also the fundamental of the next technique.

4.2 jump by two or more

After we do a local basis processing with blocksize d, the fisrt vector will be short, and it's easy to see that the next few vector is not too long also. If we make our sieving context jump to right by two index or more, say we jump s steps after each SVP subroutine, we accelerate by factor s while not degenerating the quality much.

Here a crucial point is to use a blocksize slightly larger than we require. For instance, If we want to do a BKZ with blocksize d, we can choose a $d' = d + s$, then every time we jump s steps. The result will not be worse than do BKZ-d without jump (actually better), if the output basis has the similar quality as a *HKZ-reduced* basis. After the modification, the number of SVP subroutine is only $\frac{1}{s}$ as before, and for each subroutine, the cost is $2^{0.36s}$ (practically, when $d \sim 90$) as before if we use SVP algorithm based on 3-sieve. Take $s = 4$ we get a speed up of at least $2^0.56$, There is a detailed analysis of jump based on simulation of BKZ algorithm in the next section, It shows .

4.3 reduce only when we need

in practical use, we usually don't need the whole BKZ reduced basis. What we want is just a short vector (in lattice challenge) or make the tail of the distance vector large enough such that the key can be get by a size reduction (in real attack of lattce based cryptography). We only introduce the case of lattice challenge here since the another is totally similar.

For example, if we want to do BKZ-100 on a 700 dimensional lattice. For the last 6 tour of SVP subroutine, we don't need to visit all index. In fact we just need to visit the index in $[1, 600], [1, 500], [1, 400], [1, 300], [1, 200], [1, 100]$ reespectively since doing BKZ on $[1, m]$ only relates to the first $m + 99$ vectors. This way we can save half of the time for the last 6 epoch. Notice that if we use a progressively larger blocksize, even if we jump by two or more usually we stay on one dimension for only $10 \sim 20$ tours. It at least saves constant ratio of time since currently the best SVP algorithm takes exponential time.

4.4 a large final run

To solve SVP or find a short vector in low dimension, Nguyen's extreme pruning technique first reduce the basis (preprocessing) then do a heavy extremely pruned enumeration with the hope of finding vectors short enough. In BKZ for large dimensional lattice, we can choose a much larger dimension d in last SVP subroutine (working on $[1, d]$) to get a much shorter vector, to save the time for several tours of SVP subroutine with a normal blocksize. Since one tour costs $n/s \cdot T_{svp}$, which is much larger than a SVP subroutine, this method works well in practice (If we are search for the secret key hidden in the lattice, just do a large enumeration or sieving at the tail of the basis). A detailed analysis based on simulation of BKZ is presented in next section.

5 Lattice challenges and the simulation of BKZ

5.1 ideal lattice challenges

The Ideal Lattice challenge were started at 2012. It provides many different ideal lattices with dimension up to 1024. The original goal of this challenge is to test the algorithms for finding short vector in ideal lattice. But we will treat it as high dimensional random lattices to test our BKZ techniques. For each given lattice, a vector shorter than $1.05 \cdot \mathrm{gh}(L)$ can enter SVP Hall of Fame and a vector shorter than $n \cdot \det(L)^{\frac{1}{n}}$ can enter Approximate SVP Hall of Fame. We find a vector of norm 670275 in a 656 dimensional lattice, the root Hermite Factor is about 1.010.

This challenge were first finished in the summer of 2021, we used a laptop with Intel Core i7-7500U cpu (2.70 GHz) and a none optimized c++ program which was modified several times while running. The total cost is about 700 core-hours (the infomation uploaded to the website of ideal lattice challenge is wrong). Later we optimized the program and rerun it on a Xeon Silver 4208 CPU (2.10GHz), and takes only 380 core hours. Much faster than the previous record which takes 4637 thread hours to solves a 652-dimensional approximate SVP challenge.

For the SVP subroutine, to get a reduced basis we used a variant of DeepBKZ. DeepBKZ replace the LLL in the original BKZ algorithm by DeepLLL, which allows an operation called *deep insertion*. We modified the algorithm by further check for all short vectors from a enumeration if it can be insert to some former place, and always choose the candidate that can be insert most deep. For 20 random lattice (dim = 96) with determinant 1000, we run our modified DeepBKZ for 800 seconds, and compute the average of the distance vectors, we take square root of each elements and presented it below (the Gaussian Heuristic is 2442):

[2515 2519 2491 2442 2445 2409 2378 2352 2301 2271 2255 2208 2184 2152 2113 2080 2059 2041 1981 1948 1939 1882 1871 1842 1826 1790 1761 1735 1689 1670 1624 1585 1583 1538 1521 1467 1445 1401 1356 1344 1303 1269 1249 1213 1192 1153 1128 1110 1061 1044 1010 991 955 929 899 880 865 837 819 800 760 746 728 705 684 665 652 630 618 595 580 563 541 533 518 504 491 479 466 459 448 433 417 412 399 386 380 362 360 352 345 341 332 323 321 327]

These shows our SVP algorithm performs essentially on the same order of magnitude as []'s, so the acceleration of the whole task ($4637/380 \approx 2^{3.6}$) mainly comes from the techniques in the previous section. And we did not use a highly optimized blocksize strategy which may give further speed up.

5.2 simulation and analysis

It is easy to simulate the BKZ based on local processing if we know the behavior of the SVP algorithm. Actually the only we need is the (average) distance vector of basis after run the SVP subroutine.

Algorithm 2: Simulation of BKZ algorithm

Input: a distance vector $[B_1, B_2, \cdots, B_n]$, blocksize d and an average distance vector $[D_1, \cdots, D_d]$

Output: the new distance vector after run one tour of BKZ d

```
Divide D_i's by some number to make ∏_{i=1}^{d} D_i = 1;
for s = 1, ···, n − d + 1 do
    det = (∏_{i=s}^{s+d−1} D_i)^{1/d};
    for i = 0, ···, d − 1 do
        B_{i+s} = det · D_{i+1};
```

To measure the quality of a basis, we introduce the following notation:

$$\mathrm{Pot}(L) = \prod_{i=1}^{n} B_i^{n+1-i}$$

now we can analyz the techniques in the previous section.

jump by two or more We take a reduced 700 dimensional lattice (from ideal lattice challenge) and compute its distance vector, to compare the jumping s-trategies. The determinant is $1023.35^{7}00$ and the root Hermite factor of the first vector is 1.010205, corresponds to blocksize ≈ 80. we generate the *ideal* distance vector of the HKZ-reduced basis with dimension from 80 to 94 by Algorithm 3, which is based on Gaussian Heuristic (this distance vector will be slight better than the average of real samples, since for real samples sometimes the algorithm unluckily failed to find the shortest one, but it can not find a vector shorter than the shortest vector even if we are very lucky. This will not affect the results we get since we only interested in the speed up ratio):

We run the simulation for different dimensions and different jumping steps, the result is in the following tables. We set the cost of a 80 dimensional subroutine to be 1, and the cost of $80+i$ dimensional subroutine to be $2^{0.36i}$ (based on the performance of 3-sieve in practical, dim ~ 80).

Algorithm 3: generate *ideal* distance vector

Input: dimension d and an average distance vector $[D_1, \cdots, D_60]$ for 60-dimensional HKZ-basis (get from real samples)

Output: the *ideal* distance vector

det $= 1.0$;

for $i = 1, \cdots, d-60$ **do**

  $A_i = \det^{\frac{1}{d-i+1}} \cdot \frac{\Gamma(\frac{d-i+1}{2}+1)^{\frac{1}{d-i+1}}}{\sqrt{\pi}}$;

  det $= \det / A_i$;

det $= \det^{\frac{1}{60}}$;

for $i = d-59, \cdots, d$ **do**

  $A_i = D_{i-d+60} \cdot \det$;

return $[A_1, \cdots, A_d]$

Table 1. Jumping step = 1

blocksize	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94
cost	622	797	1021	1309	1676	2149	2753	3528	4520	5792	7421	9509	12184	15611	20003
ΔPot	-408	179	788	1417	2069	2741	3434	4149	4884	5641	6418	7217	8036	8875	9736
$\frac{\Delta\text{Pot}}{\text{cost}}$	-0.66	0.22	0.77	1.08	1.23	1.27	1.25	1.18	1.08	0.97	0.86	0.76	0.66	0.57	0.49

Table 2. best jumping step for different blocksize

blocksize	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94
optimal step	1	1	2	3	4	5	6	7	8	9	9	10	12	12	13
$\frac{\Delta\text{Pot}}{\text{cost}}$	-0.66	0.22	0.94	1.81	2.62	3.28	3.79	4.11	4.24	4.26	4.15	3.97	3.72	3.45	3.15

The first table shows the cost and the change of Pot after one tour of local processing if we only jump one index each time. The result shows it's optimal to choose blocksize to be 85 (always assume we have enough RAM for sieving!) and Pot decrease 1.27 per cost. The second table shows the optimal jumping step with different blocksize. We can see if we take the blocksize to be a little bit more larger, we can decrease the Pot 4.26 per cost. The jumping technique lead to a speed up of $4.26/1.27 \approx 2^{1.75}$, more than $2^{0.56}$ mentioned in the previous section, as expected.

a large final run For a given lattice, Pot is a increase function in the root Hermite factor of the first vector in the basis, if we accept the GSA assumption. So always choosing the blocksize and jumping step such that Pot of the lattice decrease fastest is optimal in a certain sense. For the same 700 dimensional lattice, we used a brute force search to find the following optimal reduction path:

Table 3. the optimal BKZ path

num tours	blocksize	step	cost	total cost	$\|\mathbf{b}_1\|$
1	89	9	652	652	1230727
2	89	8	737	1399	1217066
3	89	8	737	2125	1203294
4	90	9	836	2962	1184848
5	90	8	836	3908	1172181
6	90	8	945	4854	1158387
7	91	9	945	5928	1140806
8	91	8	1073	7142	1128556
9	91	8	1214	8356	1115499
10	91	8	1214	9570	1103897

If we want a vector with norm 1100000 by BKZ tours, the table shows the cost is more than 9570. But a straightforward computation based on Gaussian Heuristic shows that a SVP subroutine on the first 110 vector can also do this, which only costs 1782. So the large final run saves the time of the final 8 tours of BKZ in this example.

We can search the optimal time to do a final run by a brute force search since the cost of simulation is negelectable. For instance if we want a vector of length 350000, we have the following graph

After each tour, we draw the current cost (blue points) and the total cost if we use a final run to get a vector shorter than 350000 now (red points) on the graph. It shows that if we do a final run when the length is about 388000 the total cost will be $2^{25.495}$, less than $2^{26.52}$ if we simply run the BKZ tours. We can save more than half of the time with this technique (the time to get BKZ-80 reduce is negelectable here). Note that with a sieving based SVP algorithm, this

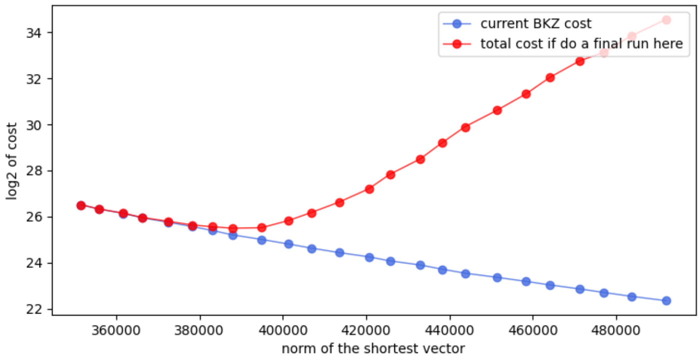

Fig. 1. part of the optimal reduction path

technique will cost more RAM. In this example, the blocksize of the final run will be 144, and we need blocksize only 129 if we only do BKZ tours, so the RAM we need grows 2^3 times. We can also get $2^{0.55}$ faster if we stop at about 366000, which requires one times more RAM. Anyway, this is totally free for enumeration based algorithms.

6 example

6.1 A Subsection Sample

Please note that the first paragraph of a section or subsection is not indented. The first paragraph that follows a table, figure, equation etc. does not need an indent, either.

Subsequent paragraphs, however, are indented.

Sample Heading (Third Level) Only two levels of headings should be numbered. Lower level headings remain unnumbered; they are formatted as run-in headings.

Sample Heading (Fourth Level) The contribution should contain no more than four levels of headings. Table 4 gives a summary of all heading levels.
Displayed equations are centered and set on a separate line.

$$x + y = z \tag{1}$$

Please try to avoid rasterized images for line-art diagrams and schemas. Whenever possible, use vector graphics instead (see Fig. 2).

Theorem 1. *This is a sample theorem. The run-in heading is set in bold, while the following text appears in italics. Definitions, lemmas, propositions, and corollaries are styled the same way.*

Table 4. Table captions should be placed above the tables.

Heading level	Example	Font size and style
Title (centered)	**Lecture Notes**	14 point, bold
1st-level heading	**1 Introduction**	12 point, bold
2nd-level heading	**2.1 Printing Area**	10 point, bold
3rd-level heading	**Run-in Heading in Bold.** Text follows	10 point, bold
4th-level heading	*Lowest Level Heading.* Text follows	10 point, italic

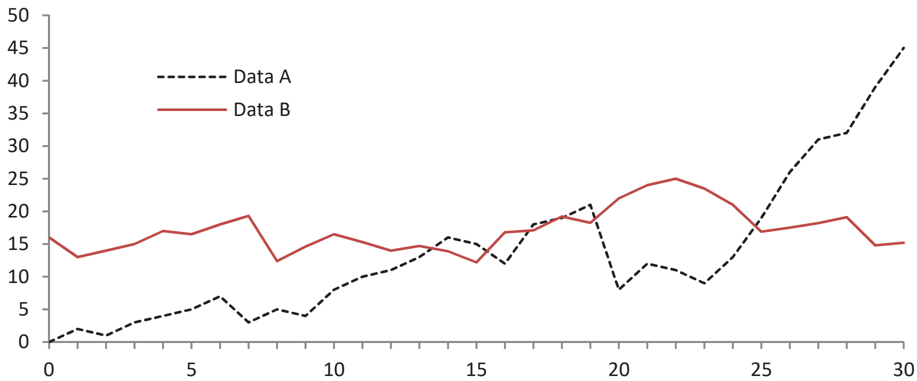

Fig. 2. A figure caption is always placed below the illustration. Please note that short captions are centered, while long ones are justified by the macro package automatically.

Proof. Proofs, examples, and remarks have the initial word in italics, while the following text appears in normal font.

For citations of references, we prefer the use of square brackets and consecutive numbers. Citations using labels or the author/year convention are also acceptable. The following bibliography provides a sample reference list with entries for journal articles [1], an LNCS chapter [2], a book [3], proceedings without editors [4], and a homepage [5]. Multiple citations are grouped [1–3], [1, 3–5].

References

1. Author, F.: Article title. Journal **2**(5), 99–110 (2016)
2. Author, F., Author, S.: Title of a proceedings paper. In: Editor, F., Editor, S. (eds.) CONFERENCE 2016, LNCS, vol. 9999, pp. 1–13. Springer, Heidelberg (2016). https://doi.org/10.10007/1234567890
3. Author, F., Author, S., Author, T.: Book title. 2nd edn. Publisher, Location (1999)
4. Author, A.-B.: Contribution title. In: 9th International Proceedings on Proceedings, pp. 1–2. Publisher, Location (2010)
5. LNCS Homepage, http://www.springer.com/lncs. Last accessed 4 Oct 2017